Madrid
Diciembre 2005

Aprenda Matlab 7.0

como si estuviera en primero

Javier García de Jalón, José Ignacio Rodríguez, Jesús Vidal



Escuela Técnica Superior
de Ingenieros Industriales
Universidad Politécnica de Madrid

Aprenda Matlab 7.0 como si estuviera en primero

Javier García de Jalón
José Ignacio Rodríguez
Jesús Vidal

ÍNDICE

1. PRÓLOGO

La colección de manuales "Aprenda Informática como si estuviera en Primero" nació en la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra) a lo largo de la década de 1990, como consecuencia de la impartición de las asignaturas Informática 1 e Informática 2, introducidas en el Plan de Estudios de 1993.

El objetivo de esta colección era facilitar a los alumnos de las asignaturas citadas unos apuntes breves y sencillos, fáciles de leer, que en unos casos ayudasen en el uso de las aplicaciones informáticas más habituales para un ingeniero industrial y en otros sirvieran de introducción a distintos lenguajes de programación.

Así pues, los destinatarios directos de estos apuntes eran los alumnos de la Escuela de Ingenieros Industriales de San Sebastián. Para facilitarles su uso, además de estar a la venta en el Servicio de Reprografía, se introdujeron versiones "online" en formato PDF (Portable Document Format, de Adobe), accesibles a través de las páginas Web de las mencionadas asignaturas. Los alumnos de cursos superiores y algunos profesores los utilizaban también para actualizar sus conocimientos cuando se instalaban nuevas versiones de las correspondientes aplicaciones.

Sin haberlos anunciado en ningún índice o buscador, al cabo de cierto tiempo se observó que eran accedidos con una frecuencia creciente desde el exterior de la Escuela, a través de Internet. Poco a poco empezaron a llegar de todo el mundo de habla hispana correos electrónicos que se interesaban por nuevos títulos, daban noticia de erratas, solicitaban permiso para utilizarlos en la docencia de otras instituciones o simplemente daban las gracias por haberlos puesto en Internet.

A la vista de estos efectos "no buscados", se estableció una página Web dedicada especialmente a esta colección y se anunció en los tres o cuatro portales más importantes de lengua española, lo que hizo que en poco tiempo se multiplicaran los accesos.

A partir del curso 2000-01 el autor principal y creador de la colección se trasladó a la Escuela Técnica Superior de Ingenieros Industriales de la Universidad Politécnica de Madrid, de la que es actualmente catedrático en el área de Matemática Aplicada. El principal punto de entrada a la colección se encuentra ahora en la dirección http://www.tayuda.com.. El número de accesos ha seguido aumentando, llegando casi a 1.000.000 de ficheros en 2004.

Aunque el mantenimiento de esta colección constituya un trabajo notable y no se saque ningún rendimiento económico de ella, da particular alegría el realizar un trabajo que tantos miles de personas consideran útil. El mantenimiento de estos manuales va a ser más difícil en los próximos años, en gran parte por el cambio en la actividad docente de su director o coordinador. Por eso serán bienvenidas todas aquellas ofertas de ayuda para mantener y crear esta colección de "Open Tutorials".

Madrid, diciembre de 2005.

Javier García de Jalón de la Fuente
(javier.garciadejalon@upm.es)

2. INTRODUCCIÓN

2.1. Acerca de este manual

Las primeras versiones de este manual estuvieron dirigidas a los alumnos de ***Informática 1*** en la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra). Esta asignatura se cursa en el primer semestre de la carrera y el aprendizaje de MATLAB constituía la primera parte de la asignatura. Se trataba pues de un manual introductorio de una aplicación que, para muchos alumnos, iba a constituir su primer contacto "profesional" con los ordenadores y/o con la programación.

Desde el curso 2000-2001, este manual se ha adaptado a la asignatura de ***Matemáticas de la Especialidad (Mecánica-Máquinas)*** (Plan 1976) y a las prácticas de ***Álgebra*** (Plan 2000) en la Escuela Técnica Superior de Ingenieros Industriales de la Universidad Politécnica de Madrid. A partir del curso 2001-02 este manual se pensó subdividir en dos: "***Aprenda MATLAB 7.0 como si estuviera en Primero***" y "***Aprenda MATLAB 7.0 como si estuviera en Segundo***", este último de carácter más avanzado[1]. En la práctica, hasta la fecha, este segundo manual nunca ha llegado a ver la luz.

Por encima de las asignaturas citadas, este manual puede ser útil a un público mucho más amplio, que incluye a alumnos de cursos superiores de las Escuelas de Ingeniería Industrial, a alumnos de Tercer Ciclo y a profesores que quieran conocer más de cerca las posibilidades que tendría MATLAB en sus asignaturas. MATLAB es una de las aplicaciones más útiles que existen para poner a punto métodos numéricos en distintas asignaturas de ingeniería. Por ser una herramienta de alto nivel, el desarrollo de programas numéricos con MATLAB puede requerir hasta un orden de magnitud menos de esfuerzo que con lenguajes de programación convencionales, como Fortran, Pascal, C/C++, Java o Visual Basic.

Se ha pretendido llegar a un equilibrio entre el detalle de las explicaciones, la amplitud de temas tratados y el número de páginas. En algunos casos, junto con las instrucciones introducidas por el usuario se incluye la salida de MATLAB; en otros casos no se incluye dicha salida, pero se espera que el lector disponga de un PC con MATLAB y vaya introduciendo esas instrucciones a la vez que avanza en estas páginas. En muchas ocasiones se anima al lector interesado a ampliar el tema con la ayuda del programa (toda la documentación de MATLAB está disponible *on-line* a través del ***Help***). En cualquier caso recuérdese que la informática moderna, más que en “saber cómo hacer algo” consiste en “saber averiguar cómo hacerlo” en pocos segundos.

2.2. Novedades en este manual

Como corresponde a la nuava versión de MATLAB que describe (la versión 7.0), este manual conprende una completa revisión del anterior, correspondiente a la versión 6.5. En este sentido, por ejemplo, todas las figuras del manual han sido renovadas, pues también el aspecto de la nueva versión de MATLAB es diferente, más en unos aspectos que en otros.

Además de la citada revisión general, en este nuevo manual se hace referencia por primera vez a ciertas novedades introducidas tanto en la versión 6.5 como en la 7.0. Las más adiciones más importantes son las siguientes:

1. Ayuda y entorno de desarrollo mejorados (Apartado 2.5, a partir de la página 9).
2. Mejoras en el debugger, que permiten establecer puntos de parada de ejecución condicional (Apartado 2.5.7, página 16).

---

[1] En realidad, el manual "Aprenda Matlab como si estuviera en Segundo" no ha llegado a ver la luz (verano de 2004). Es un viejo proyecto pendiente de disponer de tiempo para terminar su edición. Si por fin se publica, aparecerá de inmediato en http://www.tayuda.com/ayudainf

3. La posibilidad de comentar bloques de sentencias (Apartado 2.10, página 21).
4. Otros tipos de variables distintos de ***double***. Aunque MATLAB trabaja por defecto con variables ***double***, existe también la posibilidad de definir variables enteras de distinto rango, así como variables reales de simple precisión y variables lógicas (Apartado 3.3.2, página 29).
5. Función ***linsolve***, que permite optimizar la resolución de sistemas de ecuaciones lineales, lo que es quizás la tarea más utilizada de MATLAB (Apartado 4.5.4, página 46).
6. Nuevos tipos de funciones, en concreto las funciones ***inline*** (Apartado 6.4.6, página 77), las funciones anónimas (Apartado 6.4.7, página 77), y las funciones anidadas (Apartado 6.4.8, página 77).
7. Ejecución de funciuones programadas en C como librerías externas o ficheros MEX (Capítulo 7, página 98).
8. Las funciones de dibujo simplificadas en 2-D (***ezplot*** y ***ezpolar***, Apartado 8.3.3, página 116) y en 3-D (***ezplot3*** y ***ezsurf***, Apartado 9.1.1, página 122).
9. Nuevas ventanas gráficas, con muchas más posibilidades de control interactivo de las propiedades (Apartado 8.7, página 119).

2.3. El programa MATLAB

MATLAB es el nombre abreviado de "MATrix LABoratory". MATLAB es un programa para realizar cálculos numéricos con ***vectores*** y ***matrices***. Como caso particular puede también trabajar con números escalares –tanto reales como complejos–, con cadenas de caracteres y con otras estructuras de información más complejas. Una de las capacidades más atractivas es la de realizar una amplia variedad de ***gráficos*** en dos y tres dimensiones. MATLAB tiene también un lenguaje de programación propio. Este manual hace referencia a la versión 7.0 de este programa (también llamada ***release* 14**), aparecida a mediados de 2004.

MATLAB es un gran programa de cálculo técnico y científico. Para ciertas operaciones es muy rápido, cuando puede ejecutar sus funciones en código nativo con los tamaños más adecuados para aprovechar sus capacidades de vectorización. En otras aplicaciones resulta bastante más lento que el código equivalente desarrollado en C/C++ o Fortran. En la versión 6.5, MATLAB incorporó un ***acelerador JIT*** (Just In Time), que mejoraba significativamente la velocidad de ejecución de los ficheros ****.m*** en ciertas circunstancias, por ejemplo cuando no se hacen llamadas a otros ficheros ****.m***, no se utilizan estructuras y clases, etc. Aunque limitado en ese momento, cuando era aplicable mejoraba sensiblemente la velocidad, haciendo innecesarias ciertas técnicas utilizadas en versiones anteriores como la ***vectorización*** de los algoritmos. En cualquier caso, el lenguaje de programación de MATLAB siempre es una magnífica herramienta de alto nivel para desarrollar aplicaciones técnicas, fácil de utilizar y que, como ya se ha dicho, aumenta significativamente la productividad de los programadores respecto a otros entornos de desarrollo.

MATLAB dispone de un código básico y de varias librerías especializadas (***toolboxes***). En estos apuntes se hará referencia exclusiva al código básico.

MATLAB se puede arrancar como cualquier otra aplicación de ***Windows***, clicando dos veces en el icono correspondiente en el escritorio o por medio del menú ***Inicio***). Al arrancar MATLAB se abre una ventana similar a la mostrada en la Figura 1. Ésta es la vista que se obtiene eligiendo la opción ***Desktop Layout/Default***, en el menú ***View***. Como esta configuración puede ser cambiada fácilmente por el usuario, es posible que en muchos casos concretos lo que aparezca sea muy diferente. En cualquier caso, una vista similar se puede conseguir con el citado comando ***View/Desktop Layout/Default***. Esta ventana inicial requiere unas primeras explicaciones.

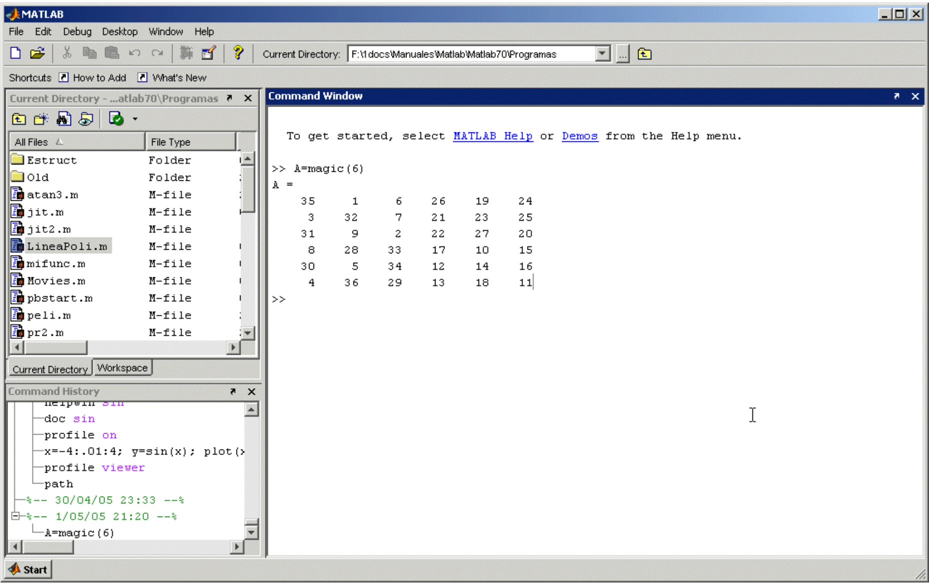


Figura 1. Ventana inicial de MATLAB 7.0.

La parte más importante de la ventana inicial es la ***Command Window***, que aparece en la parte derecha. En esta sub-ventana es donde se ejecutan los comandos de MATLAB, a continuación del ***prompt*** (aviso) característico (>>), que indica que el programa está preparado para recibir instrucciones. En la pantalla mostrada en la Figura 1 se ha ejecutado el comando **A=magic(6)**, mostrándose a continuación el resultado proporcionado por MATLAB.

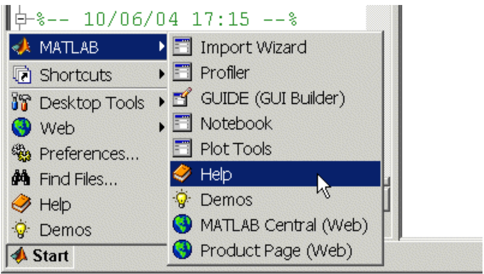


Figura 2. Menú Start/MATLAB.

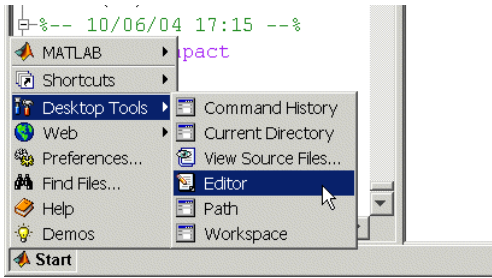


Figura 3. Menú Start/Desktop Tools.

En la parte superior izquierda de la pantalla aparecen dos ventanas también muy útiles: en la parte superior aparece la ventana ***Current Directory***, que se puede alternar con ***Workspace*** clicando en la pestaña correspondiente. La ventana ***Current Directory*** muestra los ficheros del directorio activo o actual. El directorio activo se puede cambiar desde la ***Command Window***, o desde la propia ventana (o desde la barra de herramientas, debajo de la barra de menús) con los métodos de navegación de directorios propios de ***Windows***. Clicando dos veces sobre alguno de los ficheros ****.m*** del directorio activo se abre el ***editor de ficheros*** de MATLAB, herramienta fundamental para la programación sobre la que se volverá en las próximas páginas. El ***Workspace*** contiene información sobre todas las variables que se hayan definido en esta sesión y permite ver y modificar las matrices con las que se esté trabajando.

En la parte inferior derecha aparece la ventana ***Command History*** que muestra los últimos comandos ejecutados en la ***Command Window***. Estos comandos se pueden volver a ejecutar haciendo doble clic sobre ellos. Clicando sobre un comando con el botón derecho del ratón se muestra un menú contextual con las posibilidades disponibles en ese momento. Para editar uno de estos comandos hay que copiarlo antes a la ***Command Window***.

En la parte inferior izquierda de la pantalla aparece el botón ***Start***, con una función análoga a la del botón ***Inicio*** de ***Windows***. ***Start*** da acceso inmediato a ciertas capacidades del programa. La Figura 2 muestra las posibilidades de ***Start/MATLAB***, mientras que la Figura 3 muestra las opciones de ***Start/Desktop Tools***, que permiten el acceso a las principales componentes o módulos de MATLAB. El menú ***Desktop*** realiza un papel análogo al botón ***Start***, dando acceso a los módulos o componentes de MATLAB que se tengan instalados.

Puede hacerse que al arrancar MATLAB se ejecute automáticamente un fichero, de modo que aparezca por ejemplo un saludo inicial personalizado. Esto se hace mediante un ***fichero de comandos*** que se ejecuta de modo automático cada vez que se entra en el programa (el fichero ***startup.m***, que debe estar en un directorio determinado, por ejemplo ***C:\Matlab701\Work***. Ver apartado 2.7, en la página 19).

Para apreciar desde el principio la potencia de MATLAB, se puede comenzar por escribir en la ***Command Window*** la siguiente línea, a continuación del *prompt*. Al final hay que pulsar ***intro***.

```
>> A=rand(6), B=inv(A), B*A
A =
    0.9501    0.4565    0.9218    0.4103    0.1389    0.0153
    0.2311    0.0185    0.7382    0.8936    0.2028    0.7468
    0.6068    0.8214    0.1763    0.0579    0.1987    0.4451
    0.4860    0.4447    0.4057    0.3529    0.6038    0.9318
    0.8913    0.6154    0.9355    0.8132    0.2722    0.4660
    0.7621    0.7919    0.9169    0.0099    0.1988    0.4186
B =
    5.7430    2.7510    3.6505    0.1513   -6.2170   -2.4143
   -4.4170   -2.5266   -1.4681   -0.5742    5.3399    1.5631
   -1.3917   -0.6076   -2.1058   -0.0857    1.5345    1.8561
   -1.6896   -0.7576   -0.6076   -0.3681    3.1251   -0.6001
   -3.6417   -4.6087   -4.7057    2.5299    6.1284    0.9044
    2.7183    3.3088    2.9929   -0.1943   -5.1286   -0.6537
ans =
    1.0000    0.0000         0    0.0000    0.0000   -0.0000
    0.0000    1.0000    0.0000    0.0000   -0.0000    0.0000
         0         0    1.0000   -0.0000   -0.0000    0.0000
    0.0000         0   -0.0000    1.0000   -0.0000    0.0000
   -0.0000    0.0000   -0.0000   -0.0000    1.0000    0.0000
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
```

En realidad, en la línea de comandos anterior se han escrito tres instrucciones diferentes, separadas por comas. Como consecuencia, la respuesta del programa tiene tres partes también, cada una de ellas correspondiente a una de las instrucciones. Con la primera instrucción se define una matriz cuadrada (6×6) llamada **A**, cuyos elementos son números aleatorios entre cero y uno (aunque aparezcan sólo 4 cifras, han sido calculados con 16 cifras de precisión). En la segunda instrucción se define una matriz **B** que es igual a la inversa de **A**. Finalmente se ha multiplicado **B** por **A**, y se comprueba que el resultado es la matriz unidad[2].

Es con grandes matrices o grandes sistemas de ecuaciones como MATLAB obtiene toda la potencia del ordenador. Por ejemplo, las siguientes instrucciones permiten calcular la ***potencia de cálculo del ordenador en Megaflops*** (millones de operaciones aritméticas por segundo). En la primera línea se crean tres matrices de tamaño 1000×1000, las dos primeras con valores aleatorios y la tercera con valores cero. La segunda línea toma tiempos, realiza el producto de matrices, vuelve a tomar tiempos y calcula de modo aproximado el número de millones de operaciones realizadas. La tercera lí-

[2] Al invertir la matriz y al hacer el producto posterior se han introducido pequeños errores numéricos de redondeo en el resultado, lo cual hace que no todos los elementos cero del resultado aparezcan de la misma forma.

nea calcula los Megaflops por segundo, para lo cual utiliza la función ***etime()*** que calcula el tiempo transcurrido entre dos instantes definidos por dos llamadas a la función ***clock***[3]:

```
>> n=1000; A=rand(n); B=rand(n); C=zeros(n);
>> tini=clock; C=B*A; tend=clock; mflops=(2*n^3)/1000000;
>> mflops/etime(tend,tini)
```

Otro de los puntos fuertes de MATLAB son los gráficos, que se verán con más detalle en una sección posterior. A título de ejemplo, se puede teclear la siguiente línea y pulsar ***intro***:

```
>> x=-4:.01:4; y=sin(x); plot(x,y), grid, title('Función seno(x)')
```

En la Figura 4 se puede observar que se abre una nueva ventana en la que aparece representada la función ***sin(x)***. Esta figura tiene un título "Función seno(x)" y una cuadrícula o "grid". En realidad la línea anterior contiene también varias instrucciones separadas por comas o puntos y comas. En la primera se crea un vector **x** con 801 valores reales entre -4 y 4, separados por una centésima. A continuación se crea un vector **y**, cada uno de cuyos elementos es el seno del correspondiente elemento del vector **x**. Después se dibujan los valores de **y** en ordenadas frente a los de **x** en abscisas. Las dos últimas instrucciones establecen la cuadrícula y el título.

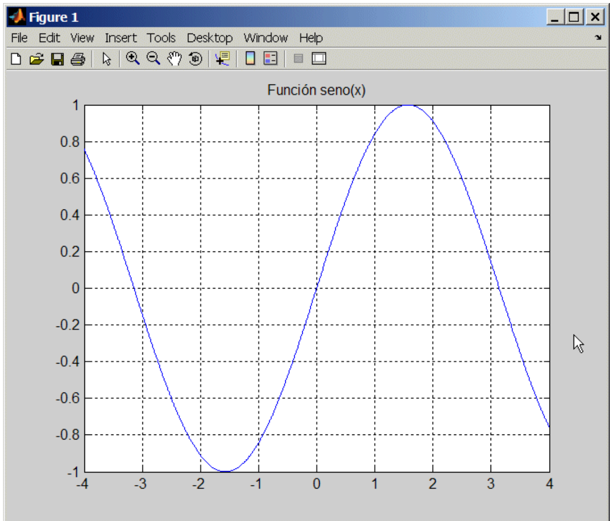


Figura 4. Gráfico de la función ***seno(x)***.

Un pequeño aviso antes de seguir adelante. Además de con la ***Command History***, es posible recuperar comandos anteriores de MATLAB y moverse por dichos comandos con el ratón y con las teclas-flechas ↑ y ↓. Al pulsar la primera de dichas flechas aparecerá el comando que se había introducido inmediatamente antes. De modo análogo es posible moverse sobre la línea de comandos con las teclas ← y →, ir al principio de la línea con la tecla ***Inicio***, al final de la línea con ***Fin***, y borrar toda la línea con ***Esc***. Recuérdese que sólo hay una línea activa (la última).

Para borrar todas las salidas anteriores de MATLAB y dejar limpia la ***Command Window*** se pueden utilizar las funciones ***clc*** y ***home***. La función ***clc*** (*clear console*) elimina todas las salidas anteriores, mientras que ***home*** las mantiene, pero lleva el ***prompt*** (**>> **) a la primera línea de la ventana.

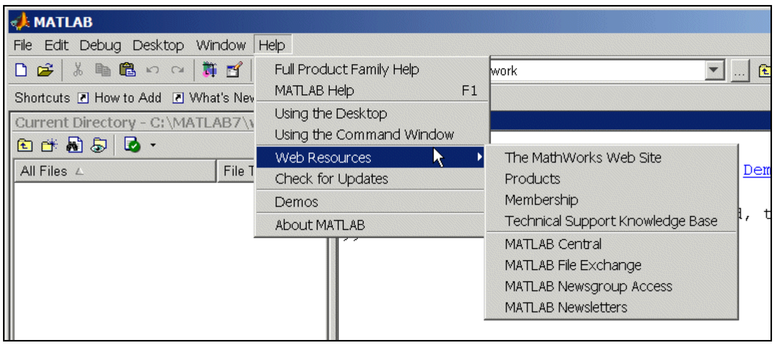


Figura 5. Menú ***Help*** de MATLAB.

Si se desea salir de MATLAB basta teclear los comandos ***quit*** o ***exit***, elegir ***Exit*** MATLAB en el menú ***File*** o utilizar cualquiera de los medios de terminar una aplicación en ***Windows***.

[3] En un portátil con Pentium IV Mobile a 2 Ghz el número de Mflops puede ser del orden de 1350. Hace 10 años un ordenador de esta potencia hubiera costado varios millones de Euros.

2.4. Uso del *Help*

MATLAB 7.0 dispone de un excelente ***Help*** con el que se puede encontrar la información que se desee. La Figura 5 muestra las distintas opciones que aparecen en el menú ***Help*** de la ventana principal de la aplicación:

1. ***Full Product Family Help***, Se abre la ventana de la Figura 8, en la que se puede buscar información general sobre MATLAB o sobre otros productos de la familia a los que se tenga acceso. La forma de la ventana de ayuda es típica y común con otros niveles de ayuda. La mayor parte de las páginas de ayuda están en formato HTML.

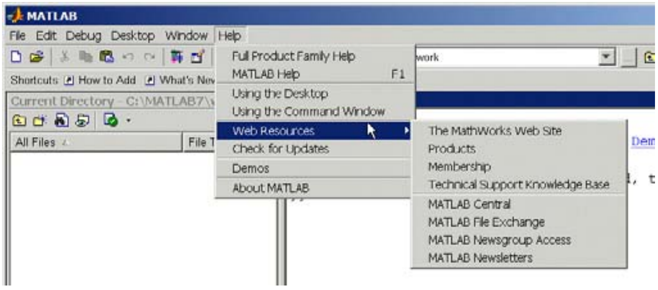


Figura 6. Algunas páginas web sobre MATLAB.

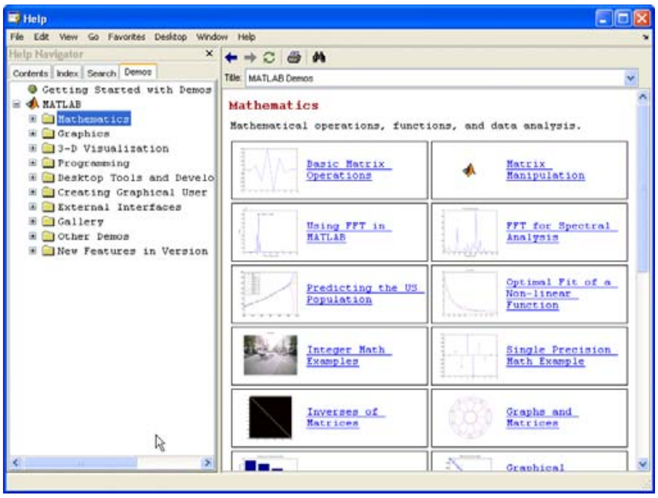


Figura 7. Demos disponibles en MATLAB.

2. ***MATLAB Help.*** Se abre la ventana de la Figura 9, en la que se puede buscar ayuda general sobre MATLAB o sobre la función o el concepto que se desee. La portada de esta ayuda tiene **tres** capítulos principales: ***Functions***, que contiene información de referencia sobre las funciones por orden alfabético o por categorías; ***Handle Graphics***, que permite acceder a información concreta sobre las distintas propiedades de los objetos gráficos; ***Documentation Set***, que da acceso a versiones completas de los manuales del programa en formato de pantalla fácilmente navegable (con apartados de *Getting Started*, *User Guides*, *Programming Tips* y *Examples in Documentation*), ***Product Demos*** (con una colección de jemplos programados que se pueden ejecutar y cuyo código se puede examinar para ver cómo están programados), ***What's New*** (con las novedades de esta versión respecto a la anterior), ***Printing the Documentation Set*** (que permite abrir documentos PDF (*Portable Document Format*), que se corrresponden con las versiones en papel de los manuales del programa, y que precisan del programa ***Adobe Acrobat Reader 5.0*** o superior.) y un apartado final sobre ***The MathWorks Web Site Resources*** (que permite acceder a una amplísima colección de informaciones adicionales disponibles en la web de la empresa que ha desarrollado MATLAB). En la parte izquierda de la ventana, cuando está seleccionada la pestaña ***Contents***, aparece un índice temático estructurado en forma de árbol que puede ser desplegado y recorrido con gran facilidad. Las restantes pestañas de esta ventana dan acceso a un índice por palabras (***Index***), a un formulario de búsqueda (***Search***) y a la colección de ejemplos ya programados antes citada (***Demos***).

3. ***Using the Desktop***. Se abre una ventana de ayuda con un formato similar a las de las Figuras anteriores con información detallada sobre cómo utilizar y configurar el entorno de desarrollo o ***Desktop***. Las distintas herramientas disponibles se describen sucesivamente. Cada página dispone de flechas y enlaces que permiten ir a la página siguiente o volver a la anterior. Es posible también imprimir aquellas páginas que se deseee consultar o archivar sobre papel. Una característica muy importante es la posibilidad de organizar las ventanas con gran flexibilidad, agrupándolas o independizándoles según los propios gustos o deseos.

4. ***Using the Command Window***. Esta opción del menú ***Help*** da acceso a la información necesaria para aprovechar las capacidades de la ***Command Window***, que es el corazón de MATLAB.
5. ***Web Resources***. La **¡Error! No se encuentra el origen de la referencia.** muestra algunas direcciones de Internet con información interesante sobre MATLAB. Todas ellas corresponden a distintas secciones de la web de The Mathworks (la empresa que desarrolla y comercializa MATLAB), cuya página de inicio se muestra en primer lugar.
6. ***Check for Updates***. MATLAB se conecta con The Mathworks y comprueba si has versiones más recientes de los productos instalados. Si se es un usuario registrado, es posible descargar las versiones más actuales.
7. ***Demos***. Se abre una ventana como la mostrada en la Figura 7 que da acceso a un buen número de ejemplos resueltos con MATLAB, cuyos resultados se presentan gráficamente de diversas formas. Es muy interesante recorrer estos ejemplos para hacerse idea de las posibilidades del programa, tanto en cálculo como en gráficos. Es asimismo muy instructivo analizar los ficheros ****.m*** de los ejemplos de características similares a las de la aplicación de se desea desarrollar.

Además, de una forma muy inmediata, es posible también recurrir al ***Help*** desde la línea de comandos de la ***Command Window***. Se aconseja practicar un poco al respecto. Por ejemplo, obsérvese la respuesta a los siguientes usos del comando ***help***:

```
>> help
>> help lang
```

El comando ***helpwin*** seguido de un nombre de comando o de función muestra la información correspondiente a ese comando en la ventana ***Help*** (ver Figura 8). En la parte superior de la ventana que se abre se muestra un enlace ***View code for …***, que permite acceder al código fuente si está disponible; con la opción ***Go to online doc for ...*** se accede a una información más completa que puede incluir ejemplos y comandos similares sobre los que también se ofrece ayuda. En la parte inferior de la página aparece una lista de enlaces ***See Also*** a funciones relacionadas.

El comando ***doc*** tecleado en la línea de comandos equivale a ***Help/Full Product Family Help***; si va seguido de un nombre de comando o función se muestra la información detallada correspondiente a ese comando de modo similar a ***Go to online doc for ...*** en el párrafo anterior.

En resumen, MATLAB dispone de una ayuda muy completa y accesible, estructurada en varios niveles (línea de comandos en la ***Command Window***, ventana ***Help***, y manuales en formato PDF), con la que es muy importante estar familiarizado, porque hasta los más expertos programadores tienen que acudir a ella con una cierta frecuencia.

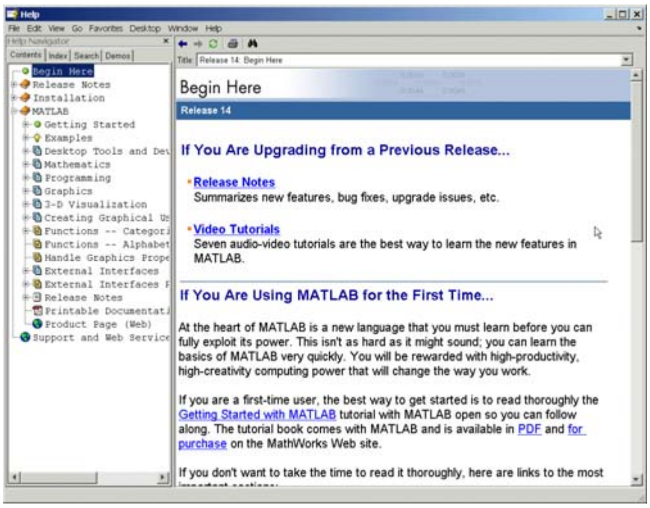


Figura 8. Ventana inicial de ***Help Full Product Family***.

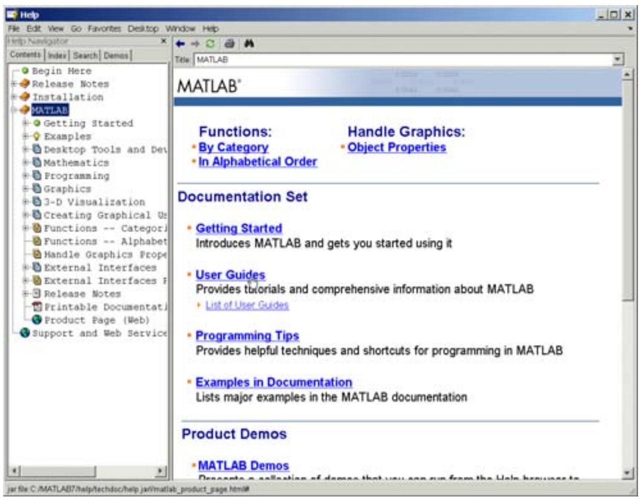


Figura 9. Ventana inicial de ***Help Matlab***.

2.5. El entorno de trabajo de MATLAB

El entorno de trabajo de MATLAB es muy gráfico e intuitivo, similar al de otras aplicaciones profesionales de ***Windows***. En la introducción a MATLAB realizada en el Apartado 2.3 y en la Figura 1, ya se han citado algunas de las componentes más importantes de este entorno de trabajo o de desarrollo. Ahora de explicarán estas componentes con un poco más de detalle.

Las componentes más importantes del entorno de trabajo de MATLAB 7.0 son las siguientes:

1. El ***Escritorio de Matlab*** (***Matlab Desktop***), que es la ventana o contenedor de máximo nivel en la que se pueden situar (*to dock*) las demás componentes.
2. Las componentes individuales, orientadas a tareas concretas, entre las que se puede citar:
   a. La ventana de comandos (***Command Window***),
   b. La ventana histórica de comandos (***Command History***),
   c. El espacio de trabajo (***Workspace***),
   d. La plataforma de lanzamiento (***Launch Pad***),
   e. El directorio actual (***Current Directory***),
   f. La ventana de ayuda (***Help***)
   g. El editor de ficheros y depurador de errores (***Editor&Debugger***),
   h. El editor de vectores y matrices (***Array Editor***).
   i. La ventana que permite estudiar cómo se emplea el tiempo de ejecución (***Profiler***).

A continuación se describen brevemente estas componentes. Téngase en cuenta que utilizar MATLAB y desarrollar programas para MATLAB es mucho más fácil si se conoce bien este entorno de trabajo. Para alcanzar la máxima productividad personal en el uso de esta aplicación es por ello muy importante leer con atención las secciones que siguen.

2.5.1. EL ESCRITORIO DE MATLAB (MATLAB DESKTOP)

El ***Matlab Desktop*** es la ventana más general de la aplicación. El resto de las ventanas o componentes citadas pueden alojarse en la ***Matlab Desktop*** o ejecutarse como ventanas independientes. A su vez, los componentes alojados en el ***Matlab Desktop*** pueden aparecer como sub-ventanas independientes o como pestañas dentro de una de las sub-ventanas. MATLAB 7.0 ofrece una gran flexibilidad al respecto y es cada usuario quien decide en qué forma desea utilizar la aplicación.

Cuando se arranca MATLAB por primera vez o cuando se ejecuta el comando ***View/Desktop Layout/Default*** aparece una ventana como la mostrada en la Figura 10. Aunque dividida en tres zonas, en realidad aparecen ***cuatro componentes***, pues la sub-ventana superior izquierda contiene dos componentes superpuestas que se permutan por medio de la pestaña correspondiente.

La Figura 11 muestra un detalle del menú ***Desktop***, desde el que se controlan las componentes visibles y la forma en que se visualizan. Por ejemplo, como en la Figura 10 la ventana activa es la ***Command Window***, en el menú de la Figura 11 aparece la opción de dejar de alojar dicha ventana en el ***Matlab Desktop*** (***Undock Command Window***). Dicho menú permite también eliminar del ***Desktop*** alguna de las componentes visibles o visualizar el ***Help*** (que no está visible). Con los submenús de ***Desktop Layout*** se pueden adoptar algunas configuraciones predefinidas, como la configuración por defecto (***Default***) o incluir sólo la ***Command Window***. La configuración adoptada por el usuario se mantendrá la siguiente vez que arranque el programa. Es posible también ***guardar*** distintas configuraciones con distintos nombres, para su uso posterior.

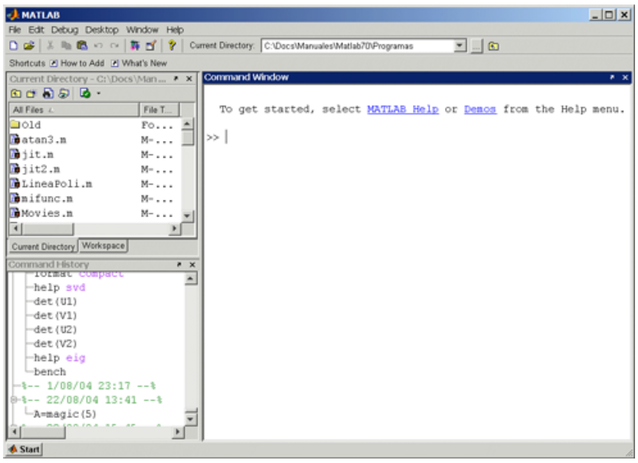


Figura 10. Configuración por defecto del *Matlab Desktop*.

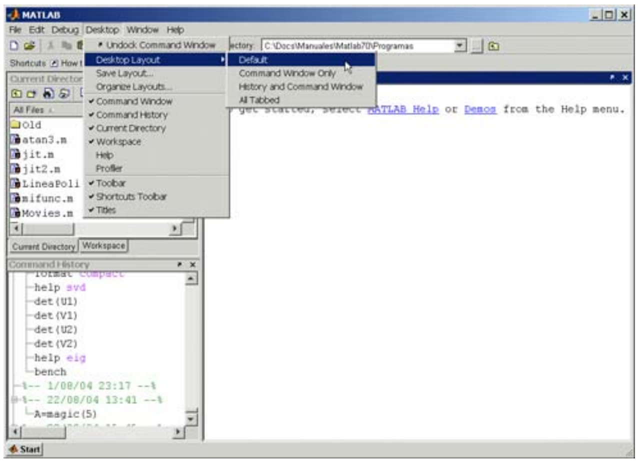


Figura 11. Menú para configurar el *Matlab Desktop*.

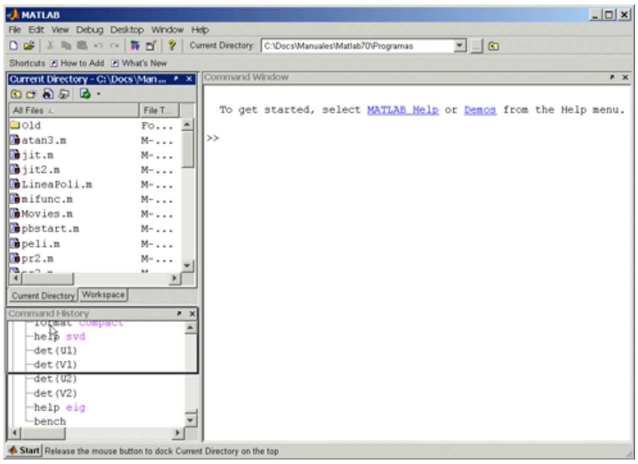


Figura 12. Arrastrar una pestaña desde una sub-ventana.

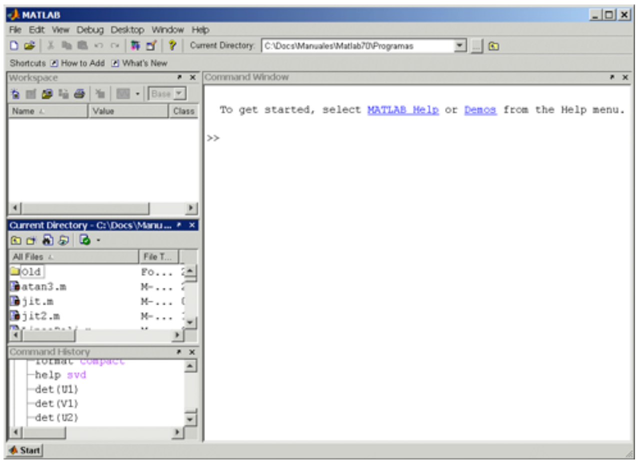


Figura 13. Creación de una nueva sub-ventana.

Además del menú mostrado en la Figura 11, que cambia en algunos detalles según cual sea la ventana activa, el usuario puede configurar el ***Matlab Desktop*** por medio del ***ratón*** mediante algunas operaciones como las siguientes:

1. Colocando el ratón sobre los bordes intermedios de las sub-ventanas y arrastrando puede modificar su tamaño en la forma que desee.
2. Clicando sobre la barra de título de la sub-ventana activa y arrastrando (Figura 12) se puede llevar a otra parte del ***Desktop***, obteniéndose el resultado mostrado en la Figura 13.
3. Si todas las ventanas se van seleccionando sucesivamente y se elige la correspondiente opción ***Undock...*** en el menú ***View***, se podría obtener una configuración como la mostrada en la Figura 14, en la que todas las ventanas son independientes y aparecen separadas en la barra de tareas.
4. Finalmente, si se parte de la configuración por defecto y cada uno de los componentes se arrastra sobre la ***Command Window*** se puede obtener una configuración como la mostrada en la Figura 15, en la que todos los componentes abiertos aparecen como pestañas alternativas en una ventana única.

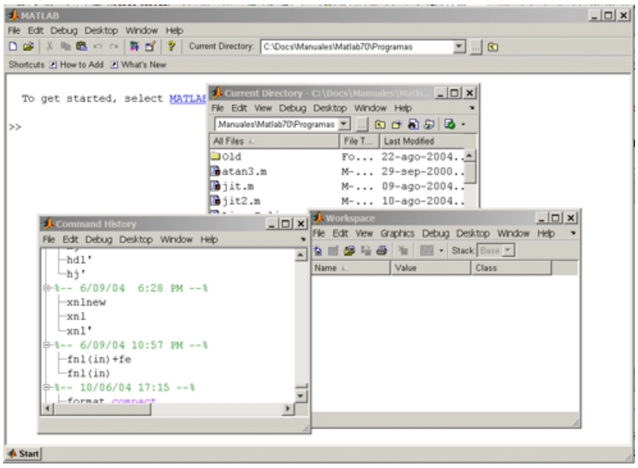


Figura 14. Ventanas independientes sobre el *Desktop*.

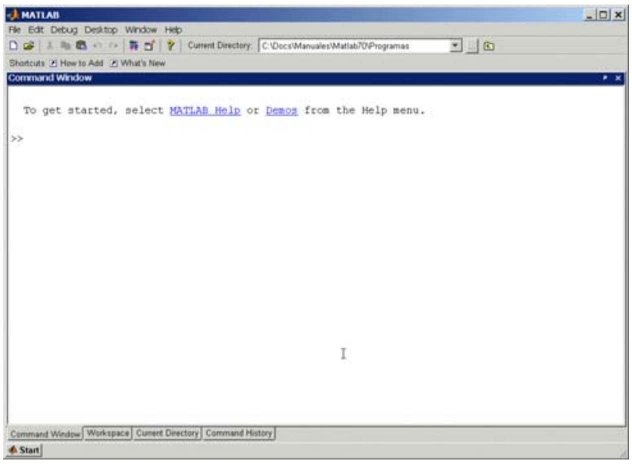


Figura 15. Todos los componentes compartiendo ventana.

La variedad de configuraciones mostradas en las figuras precedentes da una idea de las posibilidades de adaptación a las preferencias del usuario que tiene la versión 7.0 de MATLAB. Otros componentes como el ***Help Browser*** podrían añadirse a esta ventana de forma análoga. En los apartados siguientes se analizan algunas de las posibilidades de cada componente.

2.5.2. COMMAND WINDOW

Ésta es la ventana en la que se ejecutan interactivamente las instrucciones de MATLAB y en donde se muestran los resultados correspondientes, si es el caso. En cierta forma es ***la ventana más importante*** y la única que existía en las primeras versiones de la aplicación. En esta nueva versión se han añadido algunas mejoras significativas, como las siguientes:

1. Se permiten líneas de comandos muy largas que automáticamente siguen en la línea siguiente al llegar al margen derecho de la ventana. Para ello hay que activar la opción ***Wrap Lines***, en el menú ***File/Preferences/Command Window***.
2. Clicando con el botón derecho sobre el nombre de una función que aparezca en esta ventana se tiene acceso a la página del ***Help*** sobre dicha función. Si el código fuente (fichero ****.m***) está disponible, también se puede acceder al fichero correspondiente por medio del ***Editor/Debugger***.
3. Comenzando a teclear el nombre de una función y pulsando la tecla ***Tab***, MATLAB ***completa automáticamente*** el nombre de la función, o bien muestra en la línea siguiente todas las funciones disponibles que comienzan con las letras tecleadas por el usuario.
4. Cuando al ejecutar un fichero ****.m*** se produce un error y se obtiene el correspondiente mensaje en la ***Command Window***, MATLAB muestra mediante un subrayado un ***enlace a la línea del fichero fuente*** en la que se ha producido el error. Clicando en ese enlace se va a la línea correspondiente del fichero por medio del ***Editor/Debugger***.

2.5.3. COMMAND HISTORY BROWSER

La ventana ***Command History*** ofrece acceso a las sentencias que se han ejecutado anteriormente en la ***Command Window***. Estas sentencias están también accesibles por medio de las teclas ↑ y ↓ como en las versiones anteriores, pero esta ventana facilita mucho el tener una visión más general de lo hecho anteriormente y seleccionar lo que realmente se desea repetir.

Las sentencias ejecutadas anteriormente se pueden volver a ejecutar mediante un doble clic o por medio del menú contextual que se abre al clicar sobre ellas con el botón derecho. También se pue-

den copiar y volcar sobre la línea de comandos, pero se ha de copiar toda la línea, sin que se admita la copia de un fragmento de la sentencia. Existen opciones para borrar algunas o todas las líneas de esta ventana. Se puede también hacer un ***profile*** (evaluar la eficiencia relativa) de una sentencia o de un grupo de sentencias.

2.5.4. CURRENT DIRECTORY BROWSER

El concepto de ***directorio activo*** o ***directorio actual*** es muy importante en MATLAB. Los programas de MATLAB se encuentran en ficheros con la extensión ****.m***. Estos ficheros se ejecutan tecleando su nombre en la línea de comandos (sin la extensión), seguido de los argumentos entre paréntesis, si se trata de funciones. No todos los ficheros ****.m*** que se encuentren en el disco duro o en otras unidades lógicas montadas en una red local son accesibles sin más. Para que un fichero ****.m*** se pueda ejecutar es necesario que se cumpla una de las dos condiciones siguientes:

1. Que esté en el ***directorio actual***. MATLAB mantiene en todo momento un único directorio con esta condición. Este directorio es el primer sitio en el que MATLAB busca cuando desde la línea de comandos se le pide que ejecute un fichero.
2. Que esté en uno de los directorios indicados en el ***Path*** de MATLAB. El ***Path*** es una lista ordenada de directorios en los que el programa busca los ficheros o las funciones que ha de ejecutar. Muchos de los directorios del ***Path*** son propios de MATLAB, pero los usuarios también pueden añadir sus propios directorios, normalmente al principio o al final de la lista. En un próximo apartado se verá cómo se controla el ***Path***.

El comando ***pwd*** (de *print working directory*) permite saber cuál es el ***directorio actual***. Para cambiar de ***directorio actual*** se puede utilizar el comando ***cd*** (de *change directory*) en la línea de comandos, seguido del nombre del directorio, para el cual se puede utilizar un ***path*** absoluto (por ejemplo ***cd C:\Matlab\Ejemplos***) o relativo (***cd Ejemplos***). Para subir un nivel en la jerarquía de directorios se utiliza el comando ***cd ..***, y ***cd ../..*** para subir dos niveles. Éste es el mismo sistema que se sigue para cambiar de directorio en las ventanas de MS-DOS. MATLAB permite utilizar la barra normal (/) y la barra invertida (\), indistintamente.

La ventana ***Current Directory*** permite explorar los directorios del ordenador en forma análoga a la del ***Explorador*** u otras aplicaciones de ***Windows***. Cuando se llega al directorio deseado se muestran los ficheros y ficheros allí contenidos. La ventana ***Current Directory*** permite ordenarlos por fecha, tamaño, nombre, etc. El directorio actual cambia automáticamente en función del directorio seleccionado con este explorador, y también se puede cambiar desde la propia barra de herramientas del ***Matlab Desktop***. Los ficheros ****.m*** mostrados en la ventana ***Current Directory*** se pueden abrir con el ***Editor/Debugger*** mediante un doble clic.

A partir del menú contextual que se abre clicando con el botón derecho en cualquier parte de la ventana ***Current Directory*** se tiene la posibilidad de añadir ese directorio al ***Path*** de MATLAB.

2.5.5. *PATH* DE MATLAB: ESTABLECER EL CAMINO DE BÚSQUEDA (*SEARCH PATH*)

MATLAB puede llamar a una gran variedad de funciones, tanto propias como programadas por los usuarios. Puede incluso haber ***funciones distintas con el mismo nombre***. Interesa saber cuáles son las reglas que determinan qué función o qué fichero ****.m*** es el que se va a ejecutar cuando su nombre aparezca en una línea de comandos del programa. Esto queda determinado por el *camino de búsqueda* (***search path***) que el programa utiliza cuando encuentra el nombre de una función.

El ***search path*** de MATLAB es una lista de directorios que se puede ver y modificar a partir de la línea de comandos, o utilizando el cuadro de diálogo ***Set Path***, del menú ***File***. El comando ***path***

hace que se escriba el ***search path*** de MATLAB (el resultado depende de en qué directorio esté instalado MATLAB; se muestran sólo unas pocas líneas de la respuesta real del programa):

```
>> path
>> path

        MATLABPATH

    C:\MATLAB701\toolbox\matlab\general
    C:\MATLAB701\toolbox\matlab\ops
    C:\MATLAB701\toolbox\matlab\lang
    C:\MATLAB701\toolbox\matlab\elmat
    ...
    C:\MATLAB701\toolbox\matlab\helptools
    C:\MATLAB701\toolbox\matlab\winfun
    C:\MATLAB701\toolbox\matlab\demos
    C:\MATLAB701\toolbox\local
```

Para ver cómo se utiliza el ***search path*** supóngase que se utiliza la palabra ***nombre1*** en un comando. El proceso que sigue el programa para tratar de conocer qué es ***nombre1*** es el siguiente:

1. Comprueba si ***nombre1*** es una variable previamente definida por el usuario.
2. Comprueba si ***nombre1*** es una función interna o intrínseca de MATLAB.
3. Comprueba si ***nombre1*** es una *sub-función* o una función *privada* del usuario (apartado 6.3).
4. Comprueba si hay un fichero llamado ***nombre1.mex***, ***nombre1.dll*** o ***nombre1.m*** en el ***directorio actual***, cuyo contenido se obtiene con el comando ***dir***. Ya se ha visto cómo se cambiaba el ***directorio actual***.
5. Comprueba si hay ficheros llamados ***nombre1.mex***, ***nombre1.dll*** o ***nombre1.m*** en los directorios incluidos en el ***search path*** de MATLAB.

Estos pasos se realizan por el orden indicado. En cuanto se encuentra lo que se está buscando se detiene la búsqueda y se utiliza el fichero que se ha encontrado. Conviene saber que, a igualdad de nombre, los ficheros ****.mex*** tienen precedencia sobre los ficheros ****.m*** que están en el mismo directorio.

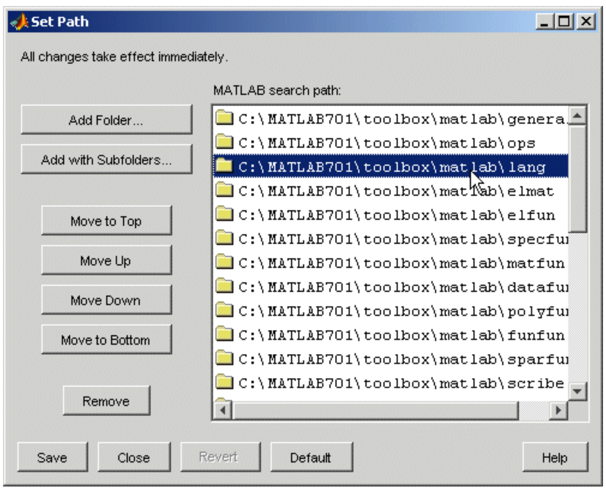


Figura 16. Cuadro de diálogo ***Set Path***.

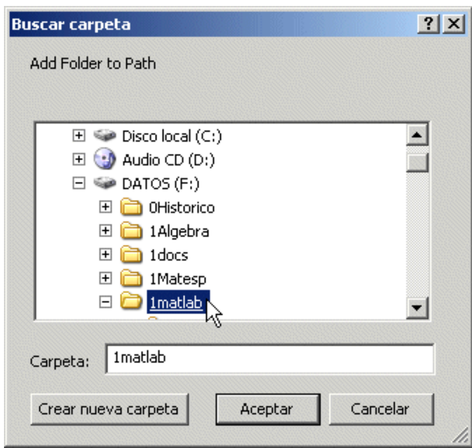


Figura 17. Añadir un directorio al ***Path***.

El cuadro de diálogo que se abre con el comando ***File/Set Path*** ayuda a definir la lista de directorios donde MATLAB debe buscar los ficheros de comandos y las funciones, tanto del sistema como de usuario. Al ejecutar dicho comando aparece el cuadro de diálogo de la Figura 16, en el cual se muestra la lista de directorios en la que MATLAB buscará. Para añadir (o quitar) un directorio a

esta lista se debe clicar sobre los botones ***Add Folder*** o ***Add with Subfolders***, con lo cual aparece un nuevo cuadro de diálogo, mostrado en la Figura 17, que ayuda a elegir el directorio deseado. El nuevo directorio se añade al comienzo de la lista, pero desde esa posición puede desplazarse hacia abajo o hasta el final con los botones ***Move Down*** o ***Move to Botton***, respectivamente. Como ya se ha dicho el orden de la lista es muy importante, porque refleja el orden de la búsqueda: si dos funciones con el mismo nombre están en dos directorios diferentes, se utilizará la que primero se encuentre. El cuadro de diálogo ***Set Path*** contiene los botones necesarios para realizar todas las operaciones que el usuario desee.

Para incluir desde la línea de comandos de MATLAB un directorio nuevo al comienzo del ***Path*** sin utilizar el cuadro de diálogo ***Set Path***, se puede utilizar también el comando ***path***, que concatena dos listas de directorios (sólo se deben utilizar directorios que realmente existan en el PC), como por ejemplo:

```
>> path('c:\mat\matlab', path)
```
[4]

mientras que para añadir el nuevo directorio al final de la lista, se utilizaría el comando:

```
>> path(path, 'c:\mat\practicas')
```

El comando ***addpath*** permite añadir uno o más directorios al ***Path***. Su forma general puede verse en los siguientes ejemplos:

```
>> addpath 'c:\Matlab' 'c:\Temp' -end
>> addpath 'c:\Matlab\Pruebas' 'c:\Temp\Pruebas' -begin
```

donde la opción por defecto (cuando no se pone ni *–begin* ni *–end*) es añadir al comienzo de la lista. Después de ejecutar estos comandos conviene comprobar cómo ha quedado modificado el ***search path*** (recuérdese que los directorios deben existir en realidad).

No es difícil borrar las líneas que se han introducido en el ***Path***: por una parte, los cambios no son permanentes y dejarán de surtir efecto al salir de MATLAB y volver a entrar (salvo que se guarden como opciones estables). Además se puede utilizar el comando ***rmpath*** (de *remove path*), al que se le pasan la lista de directorios a eliminar del ***Path***. Por ejemplo, el comando:

```
>> rmpath 'c:\Matlab' 'c:\Temp'
```

borra del ***Path*** los dos directorios indicados.

2.5.6. WORKSPACE BROWSER Y ARRAY EDITOR

El espacio de trabajo de MATLAB (***Workspace***) es el conjunto de variables y de funciones de usuario que en un determinado momento están definidas en la memoria del programa o de la función que se está jecutando. Para obtener información sobre el ***Workspace*** desde la línea de comandos se pueden utilizar los comandos ***who*** y ***whos***. El segundo proporciona una información más detallada que el primero. Por ejemplo, una salida típica del comando ***whos*** es la siguiente:

```
>> whos
  Name      Size         Bytes  Class

  A         3x3             72  double array
  B         3x3             72  double array
  C         3x3             72  double array
  D         3x3             72  double array

Grand total is 36 elements using 288 bytes
```

[4] El comando ***path*** dentro del paréntesis de la función devuelve la lista de directorios anterior.

Éstas son las variables del ***espacio de trabajo base*** (el de la línea de comandos de MATLAB). Más adelante se verá que ***cada función tiene su propio espacio de trabajo***, con variables cuyos nombres no interfieren con las variables de los otros espacios de trabajo.

La ventana ***Workspace*** constituye un entorno gráfico para ver las variables definidas en el espacio de trabajo. Se activa con el comando ***View/Workspace***. La Figura 18 muestra el aspecto inicial de la ventana ***Workspace*** cuando se abre desde un determinado programa. Haciendo doble clic por ejemplo sobre la matriz **BARS** aparece una nueva ventana (o pestaña, si la ventana ya existía) del ***Array Editor***, en la que se muestran y pueden ser modificados los elementos de dicha matriz (ver Figura 19).

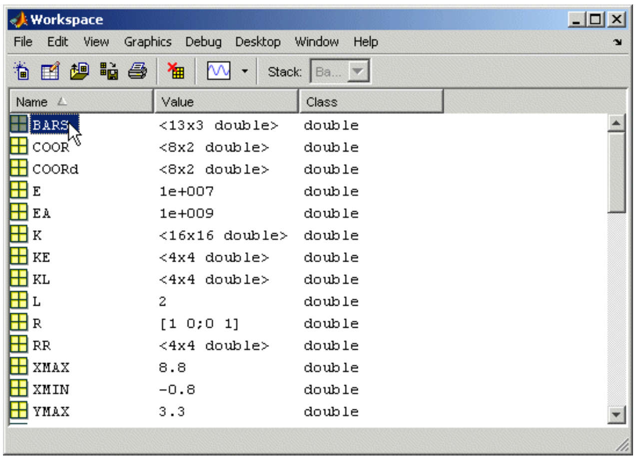


Figura 18. Workspace Browser con elementos definidos.

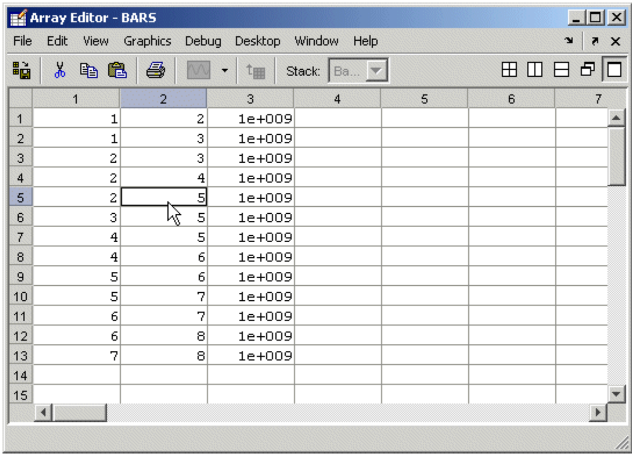


Figura 19. Array Editor (Editor de Matrices).

Es importante insistir en que cada una de las funciones de MATLAB tiene su propio espacio de trabajo, al que en principio sólo pertenecen las variables recibidas como argumentos o definidas dentro de la propia función. En la barra de herramientas de la ventana ***Workspace*** aparece una lista desplegable llamada ***Stack***, con los espacios de trabajo del programa actual. Hay que tener en cuenta que cuando se termina de ejecutar una función y se devuelve el control al programa que la había llamado, las variables definidas en la función dejan de existir (salvo que se hayan declarado como ***persistent***) y también deja de existir su espacio de trabajo.

Si se desean examinar otras matrices y/o vectores, al hacer doble clic sobre ellas el ***Array Editor*** las muestra en la misma ventana como subventanas con una pestaña diferente.

Clicando con el botón derecho sobre alguna de las variables del ***Workspace Browser*** se abre un menú contextual que ofrece algunas posibilidades interesantes, como por ejemplo la de ***representar gráficamente*** dicha variable.

El ***Array Editor*** no sólo permite ver los valores de los elementos de cualquier matriz o vector definido en el programa: es también posible ***modificar estos valores*** clicando sobre la celda correspondiente. La ventana del ***Array Editor*** incluye una lista desplegable en la que se puede elegir el formato en el que se desea ver los datos.

El ***Array Editor*** es muy útil también para entender bien ciertos algoritmos, ejecutando paso a paso un programa y viendo cómo cambian los valores de las distintas variables. Es posible aparcar o situar las ventanas o pestañas del ***Array Editor*** en la misma ventana del ***Editor/Debugger***, que se va a ver a continuación.

2.5.7. EL EDITOR/DEBUGGER

En MATLAB tienen particular importancia los ya citados ***ficheros-M*** (o ***M-files***). Son ficheros de texto ASCII, con la extensión ****.m***, que contienen ***conjuntos de comandos*** o ***definición de funciones*** (estos últimos son un poco más complicados y se verán más adelante). La importancia de estos ***ficheros-M*** es que al teclear su nombre en la línea de comandos y pulsar ***Intro***, se ejecutan uno tras otro todos los comandos contenidos en dicho fichero. El poder guardar instrucciones y grandes matrices en un fichero permite ahorrar mucho trabajo de tecleado.

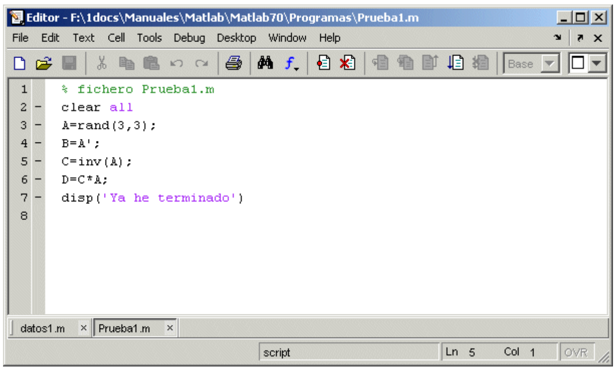


Figura 20. Ventana del Editor/Debugger.

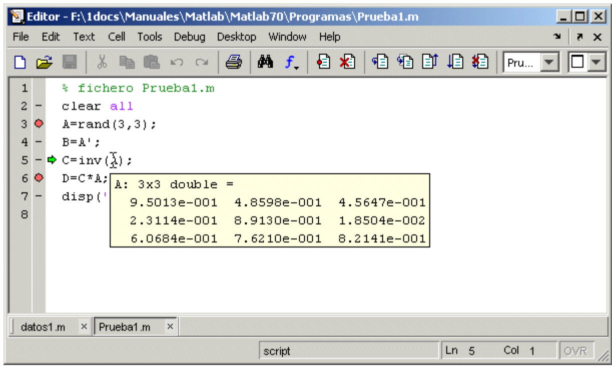


Figura 21. Ejecución interactiva con el Editor/Debugger.

Aunque los ficheros ****.m*** se pueden crear con cualquier editor de ficheros ASCII tal como ***Notepad***, MATLAB dispone de un ***editor*** que permite tanto crear y modificar estos ficheros, como ejecutarlos paso a paso para ver si contienen errores (proceso de ***Debug*** o depuración). La Figura 20 muestra la ventana principal del ***Editor/Debugger***, en la que se ha tecleado un ***fichero-M*** llamado ***Prueba1.m***, que contiene un comentario y seis sentencias[5]. El ***Editor*** muestra con diferentes colores los diferentes tipos o elementos constitutivos de los comandos (en *verde* los comentarios, en *violeta* las cadenas de caracteres, etc.). El ***Editor*** se preocupa también de que las comillas o paréntesis que se abren, no se queden sin el correspondiente elemento de cierre. Colocando el cursor antes o después de una apertura o cierre de corchete o paréntesis y pulsando las teclas (←) o (→), el ***Editor*** muestra con qué cierre o apertura de corchete o paréntesis se empareja el elemento considerado; si no se empareja con ninguno, aparece con una rayita de tachado.

Seleccionando varias líneas y clicando con el botón derecho aparece un menú contextual cuya sentencia ***Comment*** permite entre otras cosas ***comentar con el carácter %*** todas las líneas seleccionadas. Estos comentarios pueden volver a su condición de código ejecutable seleccionándolos y ejecutando ***Uncomment*** en el menú contextual. Otra opción muy útil de ese menú contextual es ***Smart Indent***, que organiza el sangrado de los bucles y bifurcaciones de las sentencias seleccionadas.

La Figura 21 corresponde a una ejecución de este fichero de comandos controlada con el ***Debugger***. Dicha ejecución se comienza eligiendo el comando ***Run*** en el menú ***Debug***, pulsando la tecla **F5**, clicando en el botón ***Continue*** () de la barra de herramientas del ***Editor*** o tecleando el nombre del fichero en la línea de comandos de la ***Command Window***. Los puntos rojos que aparecen en el margen izquierdo son ***breakpoints*** (puntos en los que se detiene la ejecución de programa); la ***flecha verde*** en el borde izquierdo indica la sentencia en que está detenida la ejecución (antes de ejecutar dicha sentencia); cuando el cursor se coloca sobre una variable (en este caso sobre **A**) aparece una pequeña ***ventana con los valores numéricos*** de esa variable, tal como se ve en la Figura 21.

---

[5] Las seis sentencias de ***prueba1.m*** son las siguientes (reagrupadas en dos líneas):

```
clear all; A=rand(3,3); B=A';
C=inv(A); D=C*A; disp('Ya he terminado');
```

En la Figura 21 puede apreciarse también que están activados los botones que corresponden al ***Debugger***. El significado de estos botones, que aparece al colocar sobre ellos el cursor, es el siguiente:

 ***Set/Clear Breakpoint***. Coloca o borra un ***breakpoint*** en la línea en que está el cursor.

 ***Clear All Breakpoints***. Elimina todos los ***breakpoints*** que haya en el fichero.

 ***Step***. Avanzar un paso sin entrar en las funciones de usuario llamadas en esa línea.

 ***Step In***. Avanzar un paso, y si en ese paso hay una llamada a una función cuyo fichero ****.m*** está accesible, entra en dicha función.

 ***Step Out***. Salir de la función que se está ejecutando en ese momento.

 ***Continue***. Continuar la ejecución hasta el siguiente ***breakpoint***.

 ***Quit Debugging***. Terminar la ejecución del ***Debugger***.

Stack. En la parte derecha de la barra de herramientas aparece esta lista desplegable (visible en la Figura 21 con las letras ***Pru...***) mediante la cual se puede elegir el *contexto*, es decir el ***espacio de trabajo*** o el ámbito de las variables que se quieren examinar. Ya se ha comentado que el espacio de trabajo base (el de las variables creadas desde la línea de comandos) y el espacio de trabajo de cada función son diferentes.

El ***Debugger*** es un programa que hay que conocer muy bien, pues es muy útil para detectar y corregir errores. Es también enormemente útil para aprender métodos numéricos y técnicas de programación. Para aprender a manejar el ***Debugger*** lo mejor es ***practicar***.

Cuando se está ejecutando un programa con el ***Debugger***, en cualquier momento se puede ir a la línea de comandos de MATLAB y teclear una expresión para ver su resultado. También se puede seleccionar con el ratón una sub-expresión en cualquier línea vista en el ***Editor/Debugger***, clicar con el botón derecho y en el menú contextual que se abre elegir ***Evaluate Selection***. El resultado de evaluar esa sub-expresión aparece en la línea de comandos de MATLAB.

Ya en las versiones anteriores MATLAB disponía de un ***Debugger alfanumérico*** que se utilizaba desde la línea de comandos y en el que está basado el nuevo ***Debugger gráfico*** del que se ha hablado anteriormente. De hecho, al realizar operaciones con el *Debugger gráfico* van apareciendo las correspondientes instrucciones en la línea de comandos de MATLAB. Para más información sobre los comandos del ***Debugger alfanumérico***, buscar en la sección "*Editing and Debugging M-Files*" en ***Help/Matlab/Desktop Tools and Development Environment***.

Seleccionando el ***nombre de una función o de un fichero de comandos*** en el código mostrado en la ventana del ***Editor/Debugger*** y abriendo con el botón derecho el menú contextual correspondiente, se ofrecen las tres posibilidades ***Evaluate Selection***, ***Open Selection*** y ***Help on Selection***, que son muy útiles para comprobar, ver o recibir ayuda sobre la función seleccionada.

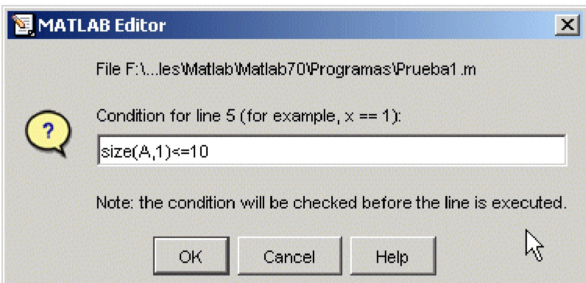


Figura 22. Establecer una condición en un breakpoint.

MATLAB permite también introducir ***breakpoints condicionales*** (indicados con un ***punto amarillo***, en vez de rojo), en los que el programa se para sólo si se cumple una determinada condición. Para introducir un ***breakpoint condicional*** basta clicar con el botón derecho en la correspondiente línea del código en la ventana del ***Editor/Debugger*** y elegir en el menú contextual que resulta ***Set/Modify Conditional Breakpoint***. Se abre una ventana como la mostrada en la Figura 22 en la que se escribe la condición que debe cumplirse para que el programa se detenga en dicho punto.

2.5.8. EL PROFILER

El ***profiler*** es un programa de utilidad que permite saber cómo se ha empleado el tiempo de la CPU en la ejecución de un determinado programa. El ***profiler*** es una herramienta muy útil para determinar los cuellos de botella de un programa, es decir las funciones y las líneas de código que más veces se llaman y que se llevan la mayor parte del tiepo de ejecución. Por ejemplo, es obvio que si se trata de mejorar la eficiencia de un programa, sería más importante mejorar una función que se llevase el 60% del tiempo total que otra que sólo se llevase el 2%. Dentro de la función más llamada, el ***profiler*** proporcina información sobre el tiempo que se lleva cada sentencia, y da también algunas orientaciones sobre las posibilidades de mejorarla.

Para explicar el ***profiler*** es mejor haber avanzado más en el conocimiento de MATLAB. Por eso, su presentación se demorará hasta la sección 6.11, a partir de la página 95.

2.6. Preferencias: Formatos de salida y de otras opciones de MATLAB

MATLAB 7.0 dispone de un cuadro de diálogo desde el que se establecen casi todas las opciones que el usuario puede determinar por su cuenta. Este cuadro de diálogo se abre con el comando ***Preferences*** del menú ***File***. En la Figura 23 se aparece el cuadro de diálogo ***Preferences*** mostrando todas las posibilidades que ofrece en el menú de la izquierda: en total son 24 cuadros de diálogo diferentes. La Figura 24 muestra el que permite elegir los colores generales del código.

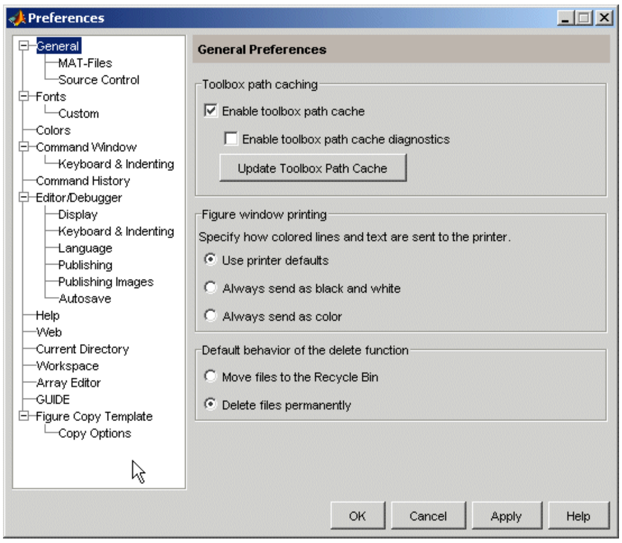


Figura 23. Cuadro de diálogo ***Preferences/General***.

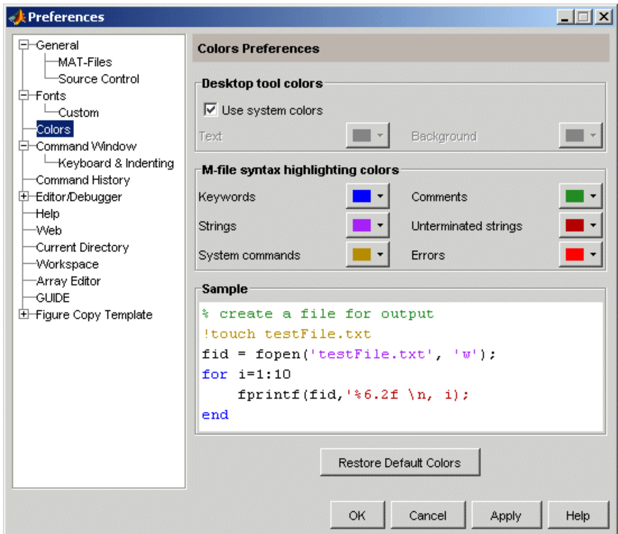


Figura 24. Cuadro de diálogo ***Preferences/ Color***.

El cuadro de diálogo ***Command Window/Fonts*** ofrece la posibilidad de elegir el tipo de letra –así como el tamaño y el color, tanto de las letras como del fondo– con la que se escribe en la ventana de comandos de MATLAB. Es muy importante utilizar tipos de letra de tamaño constante (por ejemplo, ***Courier New***, ***Lucida Console*** o ***Monospaced***), para que las filas de las matrices se alineen bien en la pantalla.

Respecto a los formatos numéricos con que MATLAB muestra los resultados (recuérdese que siempre calcula con doble precisión, es decir con unas 16 cifras decimales equivalentes), las posibilidades existentes se muestran en la lista desplegable de la Figura 25 y son las siguientes:

short	coma fija con 4 decimales (***defecto***)
long	coma fija con 15 decimales
hex	cifras hexadecimales
bank	números con dos cifras decimales
short e	notación científica con 4 decimales

short g	notación científica o decimal, dependiendo del valor
long e	notación científica con 15 decimales
long g	notación científica o decimal, dependiendo del valor
rational	expresa los números racionales como cocientes de enteros

Estos formatos se pueden cambiar también desde la línea de comandos anteponiendo la palabra ***format***. Por ejemplo, para ver las matrices en formato ***long*** habrá que ejecutar el comando:

```
>> format long
```

Por otra parte, el formato ***loose*** introduce algunas líneas en blanco en la salida (opción por defecto), mientras que el formato ***compact*** elimina las líneas en blanco citadas (es la opción recomendada en este manual). Estas opciones están disponibles en el cuadro de diálogo de la Figura 25 y se pueden también establecer desde la línea de comandos en la forma:

```
>> format compact
```

El cuadro de diálogo de la Figura 26 permite elegir un editor de programas distinto del que trae MATLAB (*built-in editor*), así como obligar a que los ficheros se abran de modo automático al ejecutarlos con el ***Debugger***.

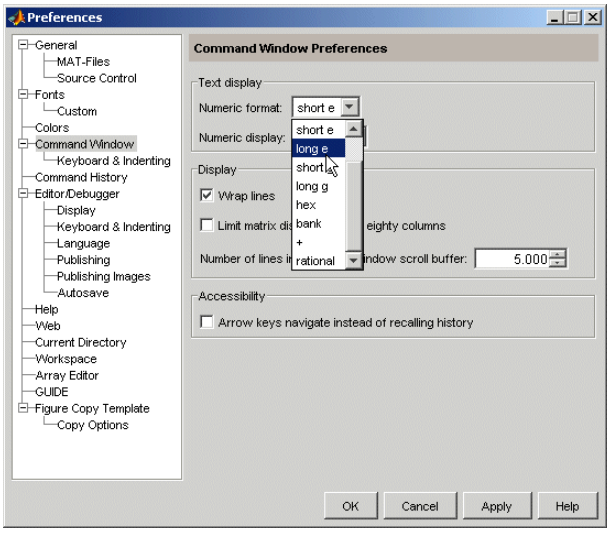


Figura 25. Cuadro de diálogo ***Prefs./Command Window***.

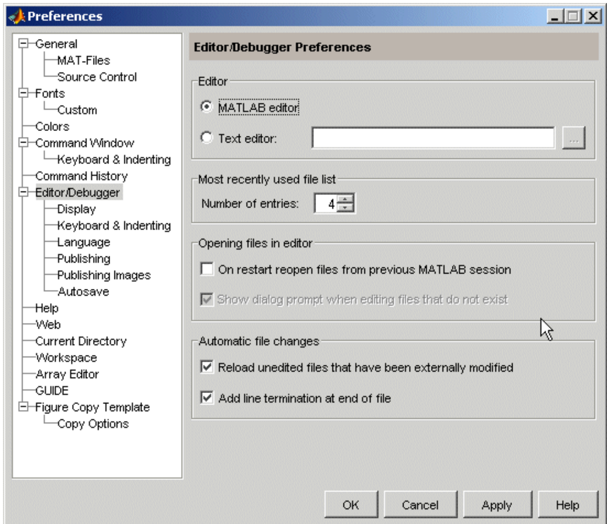


Figura 26. Cuadro de diálogo ***Prefs./Editor&Debugger***.

MATLAB aplica un ***factor de escala general*** a las matrices cuando los elementos no enteros más grandes o más pequeños son superiores o inferiores a una determinada cantidad (10^3 y 10^{-3}, respectivamente). Hay que añadir que MATLAB trata de mantener el formato de los números que han sido definidos como enteros (sin punto decimal). Si se elige la opción ***format rational*** el programa trata de expresar los números racionales como cocientes de enteros.

2.7. Ficheros matlabrc.m, startup.m y finish.m

El ***search path*** inicial o *por defecto* de MATLAB está definido en un fichero llamado ***matlabrc.m***, en el sub-directorio ***toolbox\local***. Este fichero contiene también otros parámetros de inicialización y es, por ejemplo, el responsable de los mensajes que aparecen al arrancar el programa. Este fichero se ejecuta automáticamente al arrancar MATLAB.

En las instalaciones de MATLAB en red, ***matlabrc.m*** es un fichero controlado por el administrador del sistema. Una de las cosas que hace este fichero es ver si en algún directorio del ***search path*** existe otro fichero llamado ***startup.m***, y en caso de que exista lo ejecuta. Esto abre la posibilidad de que cada usuario arranque MATLAB de una forma personalizada. Si en el ***search path*** de MA-

TLAB se coloca un fichero creado por el usuario llamado ***startup.m*** las instrucciones contenidas en dicho fichero se ejecutarán automáticamente cada vez que arranque MATLAB.

Un posible contenido de este fichero puede ser el siguiente (crearlo con el ***Editor/Debugger***):

```
>> format compact
>> addpath 'c:\Matlab\Practicas' -end
>> disp('¡Hola!')
```

Se puede crear el fichero ***startup.m*** en el directorio indicado y probar a arrancar MATLAB. Si el saludo ***¡Hola!*** se sustituye por un saludo más personal (por ejemplo, incluyendo el propio nombre), se comprobará lo explicado previamente. Es muy aconsejable crear este fichero si MATLAB se utiliza en un ordenador de uso personal.

De forma análoga, al abandonar la ejecución de MATLAB con el comando ***quit*** se ejecuta automáticamente el fichero ***finish.m***, siempre que se encuentre en alguno de los directorios del ***search path***. Este fichero se puede utilizar por ejemplo para guardar el espacio de trabajo de MATLAB (ver apartado 2.8) y poder continuar en otro momento a partir del punto en el que se abandonó el trabajo, por ejemplo al cerrar el programa.

2.8. Guardar variables y estados de una sesión: Comandos *save* y *load*

En muchas ocasiones puede resultar interesante interrumpir el trabajo con MATLAB y poderlo recuperar más tarde en el mismo punto en el que se dejó (con las mismas variables definidas, con los mismos resultados intermedios, etc.). Hay que tener en cuenta que al salir del programa todo el contenido de la memoria se borra automáticamente.

Para guardar el estado de una sesión de trabajo existe el comando ***save***. Si se teclea:

```
>> save
```

antes de abandonar el programa, se crea en el ***directorio actual*** un fichero binario llamado ***matlab.mat*** (o ***matlab***) con el estado de la sesión (excepto los gráficos, que por ocupar mucha memoria hay que guardar aparte). Dicho estado puede recuperarse la siguiente vez que se arranque el programa con el comando:

```
>> load
```

Esta es la forma más básica de los comandos ***save*** y ***load***. Se pueden guardar también matrices y vectores de forma selectiva y en ficheros con nombre especificado por el usuario. Por ejemplo, el comando (sin comas entre los nombres de variables):

```
>> save filename A x y
```

guarda las variables **A**, **x** e **y** en un fichero binario llamado ***filename.mat*** (o ***filename***). Para recuperarlas en otra sesión basta teclear:

```
>> load filename
```

Si no se indica ninguna variable, se guardan todas las variables creadas en esa sesión.

El comando ***save*** permite guardar el estado de la sesión en formato ASCII utilizándolo de la siguiente forma (lo que va detrás del carácter (%) es un comentario que es ignorado por MATLAB):

```
>> save -ascii              % almacena  8 cifras decimales
>> save -ascii -double      % almacena 16 cifras decimales
>> save -ascii -double -tab % almacena 16 cifras separadas por tabs
```

aunque en formato ASCII sólo se guardan los valores y no otra información tal como los nombres de las matrices y/o vectores. Cuando se recuperan estos ficheros con ***load -ascii*** toda la información

se guarda en una única matriz con el nombre del fichero. Esto produce un error cuando no todas las filas tienen el mismo número de elementos.

Con la opción ***-append*** en el comando ***save*** la información se guarda a continuación de lo que hubiera en el fichero.

El comando ***load*** admite las opciones ***-ascii*** y ***-mat***, para obligarle a leer en formato ASCII o binario, respectivamente.

2.9. Guardar sesión y copiar salidas: Comando *diary*

Los comandos ***save*** y ***load*** crean ficheros binarios o ASCII con el estado de la sesión. Existe otra forma más sencilla de almacenar en un fichero un texto que describa lo que el programa va haciendo (la entrada y salida de los comandos utilizados). Esto se hace con el comando ***diary*** en la forma siguiente:

```
>> diary filename.txt
...
>> diary off
...
>> diary on
...
```

El comando ***diary off*** suspende la ejecución de ***diary*** y ***diary on*** la reanuda. El simple comando ***diary*** pasa de ***on*** a ***off*** y viceversa. Para poder acceder al fichero ***filename.txt*** con ***Notepad*** o ***Word*** es necesario que ***diary*** esté en ***off***. Si en el comando ***diary*** no se incluye el nombre del fichero se utiliza por defecto un fichero llamado ***diary*** (sin extensión).

2.10. Líneas de comentarios

Ya se ha indicado que para MATLAB el carácter ***tanto por ciento*** (%) indica comienzo de comentario. Cuando aparece en una línea de comandos, el programa supone que todo lo que va desde ese carácter hasta el fin de la línea es un comentario.

Más adelante se verá que los comentarios de los ficheros ****.m*** tienen algunas peculiaridades importantes, pues pueden servir para definir ***help***'s personalizados de las funciones que el usuario vaya creando.

MATLAB permite ***comentar bloques de sentencias***, es decir, muchas sentencias contiguas de una vez. Una forma de hacerlo es seleccionar las sentencias que se desea comentar, clicar con el botón derecho, y elegir la opción ***Comment*** en el menú que se abre; las sentencias selecionadas se comentan individualmente con el carácter %. De forma similar se pueden eliminar los comentarios.

Otra forma de comentar bloques de sentencias (similar a la utilizada en C/C++ con /* y */) es encerrar las líneas que se desea inutilizar entre los caracteres **%{** y **%}**. Los bloques comentados pueden incluirse dentro de otros bloques comentados más amplios (bloques anidados).

2.11. Medida de tiempos y de esfuerzo de cálculo

MATLAB dispone de funciones que permiten calcular el tiempo empleado en las operaciones matemáticas realizadas. Algunas de estas funciones son las siguientes:

cputime — devuelve el tiempo de CPU (con precisión de centésimas de segundo) desde que el programa arrancó. Llamando antes y después de realizar una operación y restando los valores devueltos, se puede saber el tiempo de CPU empleado en esa operación. Este tiempo sigue corriendo aunque MATLAB esté inactivo.

etime(t2, t1) tiempo transcurrido entre los vectores **t1** y **t2** (¡atención al orden!), obtenidos como respuesta al comando ***clock***.

tic *ops* toc imprime el tiempo en segundos requerido por ***ops***. El comando ***tic*** pone el reloj a cero y ***toc*** obtiene el tiempo transcurrido.

A modo de ejemplo, el siguiente código mide de varias formas el tiempo necesario para resolver un sistema de 1000 ecuaciones con 1000 incógnitas. Téngase en cuenta que los tiempos pequeños (del orden de las décimas o centésimas de segundo), no se pueden medir con gran precisión.

```
>> n=1000; A=rand(n); b=rand(n,1); x=zeros(n,1);
>> tiempoIni=clock; x=A\b; tiempo=etime(clock, tiempoIni)
>> time=cputime; x=A\b; time=cputime-time
>> tic; x=A\b; toc
```

donde se han puesto varias sentencias en la misma línea para que se ejecuten todas sin tiempos muertos al pulsar ***intro***. Esto es especialmente importante en la línea de comandos en la que se quiere medir los tiempos. Todas las sentencias de cálculos matriciales van seguidas de punto y coma (;) con objeto de evitar la impresión de resultados. Conviene ejecutar dos o tres veces cada sentencia para obtener tiempos óptimos, ya que la primera vez que se ejecutan se emplea un cierto tiempo en cargar las funciones a memoria.

3. OPERACIONES CON MATRICES Y VECTORES

Ya se ha comentado que MATLAB es fundamentalmente un programa para cálculo matricial. Inicialmente se utilizará MATLAB como ***programa interactivo***, en el que se irán definiendo las matrices, los vectores y las expresiones que los combinan y obteniendo los resultados sobre la marcha. Si estos resultados son asignados a otras variables podrán ser utilizados posteriormente en otras expresiones. En este sentido MATLAB sería como una potente calculadora matricial (en realidad es esto y mucho más...).

Antes de tratar de hacer cálculos complicados, la primera tarea será aprender a introducir matrices y vectores desde el teclado. Más adelante se verán otras formas más potentes de definir matrices y vectores.

3.1. Definición de matrices desde teclado

Como en casi todos los lenguajes de programación, en MATLAB las matrices y vectores son ***variables*** que tienen ***nombres***. Ya se verá luego con más detalle las reglas que deben cumplir estos nombres. Por el momento se sugiere que se utilicen letras ***mayúsculas para matrices*** y letras ***minúsculas para vectores y escalares*** (MATLAB no exige esto, pero puede resultar útil).

Para definir una matriz *no hace falta declararlas o establecer de antemano su tamaño* (de hecho, se puede definir un tamaño y cambiarlo posteriormente). MATLAB determina el número de filas y de columnas en función del número de elementos que se proporcionan (o se utilizan). ***Las matrices se definen o introducen por filas***[6]; los elementos de una misma fila están separados por ***blancos*** o ***comas***, mientras que las filas están separadas por pulsaciones ***intro*** o por caracteres ***punto y coma*** (;). Por ejemplo, el siguiente comando define una matriz **A** de dimensión (3×3):

```
>> A=[1 2 3; 4 5 6; 7 8 9]
```

La respuesta del programa es la siguiente:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

A partir de este momento la matriz **A** está disponible para hacer cualquier tipo de operación con ella (además de valores numéricos, en la definición de una matriz o vector se pueden utilizar expresiones y funciones matemáticas). Por ejemplo, una sencilla operación con **A** es hallar su ***matriz traspuesta***. En MATLAB el apóstrofo (') es el símbolo de ***transposición matricial***. Para calcular **A'** (traspuesta de **A**) basta teclear lo siguiente (se añade a continuación la respuesta del programa):

```
>> A'
ans =
     1     4     7
     2     5     8
     3     6     9
```

Como el resultado de la operación no ha sido asignado a ninguna otra matriz, MATLAB utiliza un nombre de variable por defecto (***ans***, de *answer*), que contiene el resultado de la última operación. La variable ***ans*** puede ser utilizada como operando en la siguiente expresión que se introduzca. También podría haberse asignado el resultado a otra matriz llamada **B**:

---

[6] Aunque en MATLAB las matrices se introducen por filas, ***se almacenan por columnas***, lo cual tiene su importancia como se verá más adelante.

```
>> B=A'
B =
     1     4     7
     2     5     8
     3     6     9
```

Ahora ya están definidas las matrices **A** y **B**, y es posible seguir operando con ellas. Por ejemplo, se puede hacer el producto **B*A** (deberá resultar una matriz simétrica):

```
>> B*A
ans =
    66    78    90
    78    93   108
    90   108   126
```

En MATLAB se accede a los elementos de un vector poniendo el índice entre paréntesis (por ejemplo **x(3)** ó **x(i)**). Los elementos de las matrices se acceden poniendo los dos índices entre paréntesis, separados por una coma (por ejemplo **A(1,2)** ó **A(i,j)**). Las matrices ***se almacenan por columnas*** (aunque se introduzcan por filas, como se ha dicho antes), y teniendo en cuenta esto ***puede accederse a cualquier elemento de una matriz con un sólo subíndice***. Por ejemplo, si **A** es una matriz (3×3) se obtiene el mismo valor escribiendo **A(1,2)** que escribiendo **A(4)**.

Invertir una matriz es casi tan fácil como trasponerla. A continuación se va a definir una nueva matriz **A** -no singular- en la forma:

```
>> A=[1 4 -3; 2 1 5; -2 5 3]
A =
     1     4    -3
     2     1     5
    -2     5     3
```

Ahora se va a calcular la inversa de **A** y el resultado se asignará a **B**. Para ello basta hacer uso de la función ***inv( )*** (la precisión o número de cifras con que se muestra el resultado se puede cambiar con el menú ***File/Preferences/General***):

```
B=inv(A)
B =
    0.1803    0.2213   -0.1885
    0.1311    0.0246    0.0902
   -0.0984    0.1066    0.0574
```

Para comprobar que este resultado es correcto basta pre-multiplicar **A** por **B**;

```
>> B*A
ans =
    1.0000    0.0000    0.0000
    0.0000    1.0000    0.0000
    0.0000    0.0000    1.0000
```

De forma análoga a las matrices, es posible definir un ***vector fila*** **x** en la forma siguiente (si los tres números están separados por ***blancos*** o ***comas***, el resultado será un vector fila):

```
>> x=[10 20 30]   % vector fila
x =
    10    20    30
```

Por el contrario, si los números están separados por ***intros*** o ***puntos y coma*** (;) se obtendrá un ***vector columna***:

```
>> y=[11; 12; 13]   % vector columna
y =
    11
    12
    13
```

MATLAB tiene en cuenta la ***diferencia entre vectores fila y vectores columna***. Por ejemplo, si se intenta sumar los vectores **x** e **y** se obtendrá el siguiente mensaje de error:

```
>> x+y
??? Error using ==> +
Matrix dimensions must agree.
```

Estas dificultades desaparecen si se suma **x** con el vector transpuesto de **y**:

```
>> x+y'
ans =
    21    32    43
```

MATLAB considera ***vectores fila por defecto***, como se ve en el ejemplo siguiente:

```
>> x(1)=1, x(2)=2
x =
     1
x =
     1     2
```

A continuación se van a estudiar estos temas con un poco más de detenimiento.

3.2. Operaciones con matrices

3.2.1. OPERADORES ARITMÉTICOS

MATLAB puede operar con matrices por medio de ***operadores*** y por medio de ***funciones***. Se han visto ya los operadores *suma* (+), *producto* (*) y *traspuesta* ('), así como la función *invertir* ***inv( )***. Los operadores matriciales de MATLAB son los siguientes:

+	adición o suma
–	sustracción o resta
*	multiplicación
'	traspuesta
^	potenciación
\	división-izquierda
/	división-derecha
.*	producto elemento a elemento
./ y .\	división elemento a elemento
.^	elevar a una potencia elemento a elemento

Estos operadores se aplican también a las variables o valores escalares, aunque con algunas diferencias[7]. Todos estos operadores son coherentes con las correspondientes operaciones matriciales: no se puede por ejemplo sumar matrices que no sean del mismo tamaño. Si los operadores no se usan de modo correcto se obtiene un mensaje de error.

Los operadores anteriores se pueden aplicar también de modo ***mixto***, es decir con un operando escalar y otro matricial. En este caso la operación con el escalar se aplica a cada uno de los elementos de la matriz. Considérese el siguiente ejemplo:

[7] En términos de C++ se podría decir que son operadores *sobrecargados*, es decir, con varios significados distintos dependiendo del contexto, es decir, de sus operandos.

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
>> A*2
ans =
     2     4
     6     8
>> A-4
ans =
    -3    -2
    -1     0
```

MATLAB utiliza el ***operador de división*** / para dividir por un escalar todos los elementos de una matriz o un vector. Esto no constituye ninguna sorpresa. Sin embargo, el uso que se describe a continuación sí requiere más atención.

3.2.2. OPERADORES PARA LA RESOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES

MATLAB utiliza los ***operadores de división*** para la resolución de sistemas de ecuaciones lineales. Por su gran importancia, estos operadores requieren una explicación detenida. Considérese el siguiente sistema de ecuaciones lineales,

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

donde **x** y **b** son vectores columna, y **A** una matriz cuadrada invertible. La resolución de este sistema de ecuaciones se puede escribir en las 2 formas siguientes (¡Atención a la 2ª forma, basada en la ***barra invertida*** (\)[8], que puede resultar un poco extraña!):

$$\mathbf{x} = \mathrm{inv}(\mathbf{A})*\mathbf{b} \tag{2a}$$

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{b} \tag{2b}$$

Así pues, el operador ***división-izquierda*** por una matriz (barra invertida \) equivale a pre-multiplicar por la inversa de esa matriz. En realidad este operador es ***más general*** y ***más inteligente*** de lo que aparece en el ejemplo anterior: el operador *división-izquierda* es aplicable aunque la matriz no tenga inversa e incluso no sea cuadrada, en cuyo caso la solución que se obtiene (por lo general) es la que proporciona el *método de los mínimos cuadrados*. Cuando la matriz es ***triangular*** o ***simétrica*** aprovecha esta circunstancia para reducir el número de operaciones aritméticas. En algunos casos se obtiene una solución con no más de ***r*** elementos distintos de cero, siendo ***r*** el rango de la matriz. Esto puede estar basado en que la matriz se reduce a forma de escalón y se resuelve el sistema dando valor cero a las variables libres o independientes. Por ejemplo, considérese el siguiente ejemplo de matriz (1×2) que conduce a un sistema de infinitas soluciones:

```
>> A=[1 2], b=[2]
A =
     1     2
b =
     2
>> x=A\b
x =
     0
     1
```

que es la solución obtenida dando valor cero a la variable independiente **x(1)**. Por otra parte, en el caso de un sistema de ecuaciones ***redundante*** (o *sobre-determinado*) el resultado de MATLAB es el punto más "cercano" -en el sentido de mínima norma del error- a las ecuaciones dadas (aunque no

---

8 En inglés, MATLAB denomina ***mldivide*** a este operador. Para más información, teclear ***help mldivide***.

cumpla exactamente ninguna de ellas). Véase el siguiente ejemplo de tres ecuaciones formadas por una recta que no pasa por el origen y los dos ejes de coordenadas:

```
>> A=[1 2; 1 0; 0 1], b=[2 0 0]'
A =
     1     2
     1     0
     0     1
b =
     2
     0
     0
>> x=A\b, resto=A*x-b
x =
    0.3333
    0.6667
resto =
   -0.3333
    0.3333
    0.6667
```

Si la matriz es singular o está muy mal escalada, el operador \ da un aviso (warning), pero proporciona una solución.

La "inteligencia" del operador barra invertida \ tiene un coste: MATLAB debe de emplear cierto tiempo en determinar las características de la matriz: triangular, simétrica, etc. Si el usuario conoce perfectamente y con seguridad las características de la matriz del sistema, lo mejor es utilizar la función ***linsolve*** (ver sección 4.5.4, en la página 46), que no realiza ninguna comprobación y puede obtener la máxima eficiencia.

Aunque no es una forma demasiado habitual, también se puede escribir un sistema de ecuaciones lineales en la forma correspondiente a la traspuesta de la ecuación (1):

$$\mathbf{yB} = \mathbf{c} \tag{3}$$

donde **y** y **c** son vectores fila (**c** conocido). Si la matriz **B** es cuadrada e invertible, la solución de este sistema se puede escribir en las formas siguientes:

$$\mathbf{y} = \mathbf{c}*\text{inv}(\mathbf{B}) \tag{4a}$$

$$\mathbf{y} = \mathbf{c}/\mathbf{B} \tag{4b}$$

En este caso, el operador ***división-derecha*** por una matriz (/) equivale a postmultiplicar por la inversa de la matriz. Si se traspone la ecuación (3) y se halla la solución aplicando el operador *división-izquierda* se obtiene:

$$\mathbf{y'} = (\mathbf{B'})\backslash\mathbf{c'} \tag{5}$$

Comparando las expresiones (4b) y (5) se obtiene la relación entre los operadores *división-izquierda* y *división-derecha* (MATLAB sólo tiene implementado el operador *división-izquierda*):

$$\mathbf{c}/\mathbf{B} = ((\mathbf{B'})\backslash\mathbf{c'})' \tag{6}$$

3.2.3. Operadores elemento a elemento

En MATLAB existe también la posibilidad de aplicar ***elemento a elemento*** los operadores matriciales (*, ^, \ y /). Para ello basta precederlos por un punto (.). Por ejemplo:

```
>> [1 2 3 4]^2
??? Error using ==> ^
Matrix must be square.
```

```
>> [1 2 3 4].^2
ans =
     1     4     9    16

>> [1 2 3 4]*[1 -1 1 -1]
??? Error using ==> *
Inner matrix dimensions must agree.

>> [1 2 3 4].*[1 -1 1 -1]
ans =
     1    -2     3    -4
```

3.3. Tipos de datos

Ya se ha dicho que MATLAB es un programa preparado para trabajar con vectores y matrices. Como caso particular también trabaja con variables escalares (matrices de dimensión 1). MATLAB trabaja siempre en ***doble precisión***, es decir guardando cada dato en 8 bytes, con unas 15 cifras decimales exactas. Ya se verá más adelante que también puede trabajar con cadenas de caracteres (*strings*) y, desde la versión 5.0, también con otros tipos de datos: ***Matrices de más dos dimensiones***, ***matrices dispersas***, ***vectores y matrices de celdas***, ***estructuras*** y ***clases y objetos***. Algunos de estos tipos de datos más avanzados se verán en la última parte de este manual.

3.3.1. NÚMEROS REALES DE DOBLE PRECISIÓN

Los elementos constitutivos de vectores y matrices son números reales almacenados en 8 bytes (53 bits para la mantisa y 11 para el exponente de 2; entre 15 y 16 cifras decimales equivalentes). Es importante saber cómo trabaja MATLAB con estos números y los casos especiales que presentan.

MATLAB mantiene una forma especial para los *números muy grandes* (más grandes que los que es capaz de representar), que son considerados como ***infinito***. Por ejemplo, obsérvese cómo responde el programa al ejecutar el siguiente comando:

```
>> 1.0/0.0
Warning: Divide by zero
ans =
   Inf
```

Así pues, para MATLAB el ***infinito*** se representa como ***inf*** ó ***Inf***. MATLAB tiene también una representación especial para los resultados que no están definidos como números. Por ejemplo, ejecútense los siguientes comandos y obsérvense las respuestas obtenidas:

```
>> 0/0
Warning: Divide by zero
ans =
   NaN
>> inf/inf
ans =
   NaN
```

En ambos casos la respuesta es ***NaN***, que es la abreviatura de ***Not a Number***. Este tipo de respuesta, así como la de ***Inf***, son enormemente importantes en MATLAB, pues permiten controlar la fiabilidad de los resultados de los cálculos matriciales. Los ***NaN*** se propagan al realizar con ellos cualquier operación aritmética, en el sentido de que, por ejemplo, cualquier número sumado a un ***NaN*** da otro ***NaN***. MATLAB tiene esto en cuenta. Algo parecido sucede con los ***Inf***.

MATLAB dispone de tres funciones útiles relacionadas con las operaciones de coma flotante. Estas funciones, que no tienen argumentos, son las siguientes:

eps devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. Da una idea de la precisión o número de cifras almacenadas. En un PC, ***eps*** vale 2.2204e-016.

realmin devuelve el número más pequeño con que se puede trabajar (2.2251e-308)

realmax devuelve el número más grande con que se puede trabajar (1.7977e+308)

3.3.2. OTROS TIPOS DE VARIABLES: INTEGER, FLOAT Y LOGICAL

Como ya se ha comentado, por defecto MATLAB trabaja con variables de punto flotante y doble precisión (double). Con estas variables pueden resolverse casi todos los problemas prácticos y con frecuencia no es necesario complicarse la vida declarando variables de tipos distintos, como se hace con cualquier otro lenguaje de programación. Sin embargo, en algunos casos es conveniente declarar variables de otros tipos porque puede ahorrarse mucha memoria y pueden hacerse los cálculos mucho más rápidamente.

MATLAB permite crear variables enteras con 1, 2, 4 y 8 bytes (8, 16, 32 y 64 bits). A su vez, estas variables pueden tener signo o no tenerlo. Las variables con signo representan números en intervalos "casi" simétricos respecto al 0; las variables sin signo representan número no negativos, desde el 0 al número máximo.

Los tipos de los enteros con signo son ***int8***, ***int16***, ***int32*** e ***int64***, y sin signo ***uint8***, ***uint16***, ***uint32*** y ***uint64***. Para crear una variable entera de un tipo determinado se pueden utilizar sentencias como las siguientes:

```
>> i=int32(100);    % se crea un entero de 4 bytes con valor 100
>> j=zeros(100); i=int32(j);    % se crea un entero i a partir de j
>> i=zeros(1000,1000,'int32');  % se crea una mariz 1000x1000 de enteros
```

Las funciones **intmin('int64')** e **intmax('int64')** permiten por ejemplo saber el valor del entero más pequeño y más grande (en valor algebraico) que puede formarse con variables enteras de 64 bits:

```
>> disp([intmin('int64'), intmax('int64')])
 -9223372036854775808  9223372036854775807
```

La función **isinteger(i)** devuelve 1 si la variable i es entera y 0 en otro caso. La función **class(i)** devuelve el tipo de variable que es **i** (***int8***, ***int16***, ...), mientras que la función **isa(i, 'int16')** permite saber exactamente si la variable **i** corresponde a un entero de 16 bits.

MATLAB dispone de dos tipos de variables reales o *float*: ***single*** y ***double***, que ocupan respectivamente 4 y 8 bytes. Por defecto se utilizan ***doubles***. Las funciones **single(x)** y **double(y)** permiten realizar conversiones entre ambos tipos de variables.

Las funciones **realmin** y **realmax** permiten saber los números ***double*** más pequeño y más grande (en valor absoluto) que admite el computador. Para los correspondientes números de simple precisión habría que utilizar **realmin('single')** y **realmax('single')**. La función **isfloat(x)** permite saber si **x** es una variable real, de simple o doble precisión. Para saber exactamente de qué tipo de variable se rata se pueden utilizar las funciones **isa(x, 'single')** ó **isa(x, 'double')**. Obsérvese el ejemplo siguiente, en el que se ve cómo con variables ***single*** se reduce el tiempo de CPU y la memoria:

```
>> n=1000; AA=rand(n); A=single(AA);
>> tic, Bs=inv(A); toc
Elapsed time is 1.985000 seconds.
>> tic, Bd=inv(AA); toc
Elapsed time is 4.296000 seconds.
```

Quizás las variables más interesantes –aparte de las variables por defecto, las ***double***– sean las variables logical, que sólo pueden tomar los valores ***true*** (1) y ***false*** (0). Las variables lógicas surgen

como resultado de los operadores relacionales (==, <, <=, >, >=, ~=, ver Apartado 3.6, en la página 38) y de muchas funciones lógicas como **any** y **all** que se aplican a vectores y matrices, y que se verán en el Apartado 4.6, a partir de la página 46.

La función **logical(A)** produce una variable lógica, con el mismo número de elementos que **A**, con valores 1 ó 0 según el correspondiente elementos de **A** sea distinto de cero o igual a cero.

Una de las aplicaciones más importantes de las variables lógicas es para separar o extraer los elemntos de una matriz o vector que cumplen cierta condición, y operar luego selectivamente sobre dichos elementos. Obsérvese, el siguiente ejemplo:

```
>> A=magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> j=A>10
j =
     1     0     0     1
     0     1     0     0
     0     0     0     1
     0     1     1     0
>> isa(j,'logical')
ans =
     1
>> A(j)=-10
A =
   -10     2     3   -10
     5   -10    10     8
     9     7     6   -10
     4   -10   -10     1
```

3.3.3. NÚMEROS COMPLEJOS: FUNCIÓN *COMPLEX*

En muchos cálculos matriciales los datos y/o los resultados no son reales sino ***complejos***, con ***parte real*** y ***parte imaginaria***. MATLAB trabaja sin ninguna dificultad con números complejos. Para ver como se representan por defecto los números complejos, ejecútense los siguientes comandos:

```
>> a=sqrt(-4)
a =
        0 + 2.0000i
>> 3 + 4j
ans =
   3.0000 + 4.0000i
```

En la entrada de datos de MATLAB se pueden utilizar indistintamente la **i** y la **j** para representar el *número imaginario unidad* (en la salida, sin embargo, puede verse que siempre aparece la **i**). Si la **i** o la **j** no están definidas como variables, puede intercalarse el signo (*). Esto no es posible en el caso de que sí estén definidas, porque entonces se utiliza el valor de la variable. En general, cuando se está trabajando con números complejos, conviene no utilizar la **i** como variable ordinaria, pues puede dar lugar a errores y confusiones. Por ejemplo, obsérvense los siguientes resultados:

```
>> i=2
i =
     2
>> 2+3i
ans =
   2.0000 + 3.0000i
>> 2+3*i
```

```
ans =
     8
>> 2+3*j
ans =
   2.0000 + 3.0000i
```

Cuando **i** y **j** son variables utilizadas para otras finalidades, como *unidad imaginaria* puede utilizarse también la función **sqrt(-1)**, o una variable a la que se haya asignado el resultado de esta función.

La asignación de *valores complejos* a vectores y matrices desde teclado puede hacerse de las dos formas, que se muestran en el ejemplo siguiente (conviene hacer antes ***clear i***, para que **i** no esté definida como variable; este comando se estudiará más adelante):

```
>> A = [1+2i 2+3i; -1+i 2-3i]
A =
   1.0000 + 2.0000i   2.0000 + 3.0000i
  -1.0000 + 1.0000i   2.0000 - 3.0000i
>> A = [1 2; -1 2] + [2 3; 1 -3]*I   % En este caso el * es necesario
A =
   1.0000 + 2.0000i   2.0000 + 3.0000i
  -1.0000 + 1.0000i   2.0000 - 3.0000i
```

Puede verse que es posible definir las partes reales e imaginarias por separado. En este caso sí es necesario utilizar el operador (*), según se muestra en el ejemplo anterior.

MATLAB dispone asimismo de la función ***complex***, que crea un número complejo a partir de dos argumentos que representan la parte real e imaginaria, como en el ejemplo siguiente:

```
>> complex(1,2)
ans =
   1.0000 + 2.0000i
```

Es importante advertir que el *operador de matriz traspuesta* ('), aplicado a matrices complejas, produce la ***matriz conjugada y traspuesta***. Existe una función que permite hallar la matriz conjugada (***conj( )***) y el operador punto y apóstrofo (**.**') que calcula simplemente la matriz traspuesta.

3.3.4. CADENAS DE CARACTERES

MATLAB puede definir variables que contengan cadenas de caracteres. En MATLAB las cadenas de texto van entre apóstrofos o comillas simples (Nótese que en C van entre comillas dobles: "cadena"). Por ejemplo, en MATLAB:

```
s = 'cadena de caracteres'
```

Las cadenas de texto tienen su más clara utilidad en temas que se verán más adelante y por eso se difiere hasta entonces una explicación más detallada.

3.4. Variables y expresiones matriciales

Ya han aparecido algunos ejemplos de ***variables*** y ***expresiones*** matriciales. Ahora se va a tratar de generalizar un poco lo visto hasta ahora.

Una ***variable*** es un nombre que se da a una entidad numérica, que puede ser una matriz, un vector o un escalar. El valor de esa variable, e incluso el tipo de entidad numérica que representa, puede cambiar a lo largo de una sesión de MATLAB o a lo largo de la ejecución de un programa. La forma más normal de cambiar el valor de una variable es colocándola a la izquierda del ***operador de asignación*** (=).

Una expresión de MATLAB puede tener las dos formas siguientes: primero, asignando su resultado a una variable,

```
variable = expresión
```

y segundo evaluando simplemente el resultado del siguiente modo,

```
expresión
```

en cuyo caso el resultado se asigna automáticamente a una variable interna de MATLAB llamada ***ans*** (de *answer*) que almacena el último resultado obtenido. Se considera por defecto que una expresión termina cuando se pulsa ***intro***. Si se desea que una expresión continúe en la línea siguiente, hay que introducir ***tres puntos*** (...) antes de pulsar ***intro***. También se pueden incluir varias expresiones en una misma línea separándolas por *comas* (,) o *puntos y comas* (;).

Si una expresión ***termina en punto y coma*** (;) su resultado se calcula, pero no se escribe en pantalla. Esta posibilidad es muy interesante, tanto para evitar la escritura de resultados intermedios, como para evitar la impresión de grandes cantidades de números cuando se trabaja con matrices de gran tamaño.

A semejanza de C, ***MATLAB distingue entre mayúsculas y minúsculas*** en los nombres de variables. Los ***nombres de variables*** deben empezar siempre por una letra y pueden constar de hasta 63 letras y números. La función ***namelengthmax*** permite preguntar al programa por este número máximo de caracteres. El carácter guión bajo (_) se considera como una letra. A diferencia del lenguaje C, no hace falta declarar las variables que se vayan a utilizar. Esto hace que se deba tener especial cuidado con no utilizar nombres erróneos en las variables, porque no se recibirá ningún aviso del ordenador.

Cuando se quiere tener una *relación de las variables* que se han utilizado en una sesión de trabajo se puede utilizar el comando ***who***. Existe otro comando llamado ***whos*** que proporciona además información sobre el tamaño, la cantidad de memoria ocupada y el carácter real o complejo de cada variable. Se sugiere utilizar de vez en cuando estos comandos en la sesión de MATLAB que se tiene abierta. Esta misma información se puede obtener gráficamente con el ***Workspace Browser***, que aparece con el comando ***View/Workspace*** o activando la ventana correspondiente si estaba abierto.

El comando ***clear*** tiene varias formas posibles:

clear	sin argumentos, ***clear*** elimina todas las variables creadas previamente (excepto las variables globales).
clear A, b	borra las variables indicadas.
clear global	borra las variables globales.
clear functions	borra las funciones.
clear all	borra todas las variables, incluyendo las globales, y las funciones.

3.5. Otras formas de definir matrices

MATLAB dispone de varias formas de definir matrices. El introducirlas por teclado sólo es práctico en casos de pequeño tamaño y cuando no hay que repetir esa operación muchas veces. Recuérdese que en MATLAB no hace falta definir el tamaño de una matriz. Las matrices toman tamaño al ser definidas y este tamaño puede ser modificado por el usuario mediante adición y/o borrado de filas y columnas. A continuación se van a ver otras formas más potentes y generales de definir y/o modificar matrices.

3.5.1. Tipos de matrices predefinidos

Existen en MATLAB varias funciones orientadas a definir con gran facilidad matrices de tipos particulares. Algunas de estas funciones son las siguientes:

eye(4)	forma la matriz unidad de tamaño (4×4)

zeros(3,5)	forma una matriz de *ceros* de tamaño (3×5)
zeros(4)	ídem de tamaño (4×4)
ones(3)	forma una matriz de *unos* de tamaño (3×3)
ones(2,4)	idem de tamaño (2×4)
linspace(x1,x2,n)	genera un vector con **n** valores igualmente espaciados entre **x1** y **x2**
logspace(d1,d2,n)	genera un vector con **n** valores espaciados logarítmicamente entre 10^d1 y 10^d2. Si **d2** es **pi**[9], los puntos se generan entre 10^d1 y **pi**
rand(3)	forma una matriz de números aleatorios entre 0 y 1, con distribución uniforme, de tamaño (3×3)
rand(2,5)	idem de tamaño (2×5)
randn(4)	forma una matriz de números aleatorios de tamaño (4×4), con distribución normal, de valor medio 0 y varianza 1.
magic(4)	crea una matriz (4×4) con los números 1, 2, ... 4*4, con la propiedad de que todas las filas y columnas suman lo mismo
hilb(5)	crea una matriz de Hilbert de tamaño (5×5). La matriz de Hilbert es una matriz cuyos elementos (i,j) responden a la expresión (1/(i+j-1)). Esta es una matriz especialmente difícil de manejar por los grandes errores numéricos a los que conduce
invhilb(5)	crea directamente la inversa de la matriz de Hilbert
kron(x,y)	produce una matriz con todos los productos de los elementos del vector **x** por los elementos del vector **y**. Equivalente a **x'*y**, donde **x** e **y** son vectores fila
compan(pol)	construye una matriz cuyo polinomio característico tiene como coeficientes los elementos del vector **pol** (ordenados de mayor grado a menor)
vander(v)	construye la matriz de ***Vandermonde*** a partir del vector **v** (las columnas son las potencias de los elementos de dicho vector)

Existen otras funciones para crear matrices de tipos particulares. Con ***Help/Matlab Help*** se puede obtener información sobre todas las funciones disponibles en MATLAB, que aparecen agrupadas por categorías o por orden alfabético. En la categoría ***Mathematics*** aparecen la mayor parte de las funciones estudiadas en este apartado.

3.5.2. FORMACIÓN DE UNA MATRIZ A PARTIR DE OTRAS

MATLAB ofrece también la posibilidad de crear una matriz a partir de matrices previas ya definidas, por varios posibles caminos:

- recibiendo alguna de sus propiedades (como por ejemplo el tamaño),
- por composición de varias submatrices más pequeñas,
- modificándola de alguna forma.

A continuación se describen algunas de las funciones que crean una nueva matriz a partir de otra o de otras, comenzando por dos funciones auxiliares:

[9] **pi** es una variable predefinida en MATLAB, que como es fácil suponer representa el número π.

[m,n]=size(A)	devuelve el número de filas y de columnas de la matriz **A**. Si la matriz es cuadrada basta recoger el primer valor de retorno
n=length(x)	calcula el número de elementos de un vector **x**
zeros(size(A))	forma una matriz de *ceros* del mismo tamaño que una matriz **A** previamente creada
ones(size(A))	ídem con *unos*
A=diag(x)	forma una matriz diagonal **A** cuyos elementos diagonales son los elementos de un vector ya existente **x**
x=diag(A)	forma un vector **x** a partir de los elementos de la diagonal de una matriz ya existente **A**
diag(diag(A))	crea una matriz diagonal a partir de la diagonal de la matriz **A**
blkdiag(A,B)	crea una matriz diagonal de submatrices a partir de las matrices que se le pasan como argumentos
triu(A)	forma una matriz triangular superior a partir de una matriz **A** (no tiene por qué ser cuadrada). Con un segundo argumento puede controlarse que se mantengan o eliminen más diagonales por encima o debajo de la diagonal principal.
tril(A)	ídem con una matriz triangular inferior
rot90(A,k)	Gira k*90 grados la matriz rectangular **A** en sentido antihorario. k es un entero que puede ser negativo. Si se omite, se supone k=1
flipud(A)	halla la matriz simétrica de **A** respecto de un eje horizontal
fliplr(A)	halla la matriz simétrica de **A** respecto de un eje vertical
reshape(A,m,n)	Cambia el tamaño de la matriz **A** devolviendo una matriz de tamaño $m \times n$ cuyas columnas se obtienen a partir de un vector formado por las columnas de **A** puestas una a continuación de otra. Si la matriz **A** tiene menos de $m \times n$ elementos se produce un error.

Un caso especialmente interesante es el de crear una nueva matriz ***componiendo como submatrices*** otras matrices definidas previamente. A modo de ejemplo, ejecútense las siguientes líneas de comandos y obsérvense los resultados obtenidos:

```
>> A=rand(3)
>> B=diag(diag(A))
>> C=[A, eye(3); zeros(3), B]
```

En el ejemplo anterior, la matriz **C** de tamaño (6×6) se forma por composición de cuatro matrices de tamaño (3×3). Al igual que con simples escalares, las submatrices que forman una fila se separan con ***blancos*** o ***comas***, mientras que las diferentes filas se separan entre sí con ***intros*** o ***puntos y comas***. Los tamaños de las submatrices deben de ser coherentes.

3.5.3. DIRECCIONAMIENTO DE VECTORES Y MATRICES A PARTIR DE VECTORES

Los elementos de un vector **x** se pueden direccionar a partir de los de otro vector **v**. En este caso, **x**(**v**) equivale al vector **x(v(1))**, **x(v(2))**, ... Considérese el siguiente ejemplo:

```
>> v=[1 3 4]
v =
     1     3     4
>> x=rand(1,6)
x =
    0.5899    0.4987    0.7351    0.9231    0.1449    0.9719
>> x(v)
ans =
    0.5899    0.7351    0.9231
```

De forma análoga, los elementos de una matriz **A** pueden direccionarse a partir de los elementos de dos vectores **f** y **c**. Véase por ejemplo:

```
>> f=[2 4]; c=[1 2];
>> A=magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> A(f,c)
ans =
     5    11
     4    14
```

El siguiente ejemplo –continuación del anterior– permite comprobar cómo los elementos de una matriz se pueden direccionar con un sólo índice, considerando que las columnas de la matriz están una a continuación de otra formando un vector:

```
>> f=[1 3 5 7];
>> A(f), A(5), A(6)
ans =
    16     9     2     7
ans =
     2
ans =
    11
```

Más adelante se verá que esta forma de extraer elementos de un vector y/o de una matriz tiene abundantes aplicaciones, por ejemplo la de modificar selectivamente esos elementos.

3.5.4. OPERADOR DOS PUNTOS (:)

Este operador es muy importante en MATLAB y puede usarse de varias formas. Se sugiere al lector que practique mucho sobre los ejemplos contenidos en este apartado, introduciendo todas las modificaciones que se le ocurran y haciendo pruebas abundantes (¡Probar es la mejor forma de aprender!).

Para empezar, defínase un vector **x** con el siguiente comando:

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
```

En cierta forma se podría decir que el operador (:) representa un ***rango***: en este caso, los números enteros entre el 1 y el 10. Por defecto el incremento es 1, pero este operador puede también utilizarse con otros valores enteros y reales, positivos o negativos. En este caso el incremento va entre el valor inferior y el superior, en las formas que se muestran a continuación:

```
>> x=1:2:10
x =
     1     3     5     7     9
>> x=1:1.5:10
x =
    1.0000    2.5000    4.0000    5.5000    7.0000    8.5000   10.0000
>> x=10:-1:1
x =
    10     9     8     7     6     5     4     3     2     1
```

Puede verse que, por defecto, este operador produce vectores fila. Si se desea obtener un vector columna basta trasponer el resultado. El siguiente ejemplo genera una tabla de funciones *seno* y

coseno. Ejecútese y obsérvese el resultado (recuérdese que con (;) después de un comando el resultado no aparece en pantalla).

```
>> x=[0.0:pi/50:2*pi]';
>> y=sin(x); z=cos(x);
>> [x y z]
```

El operador dos puntos (:) es aún más útil y potente –y también más complicado– con matrices. A continuación se va a definir una matriz **A** de tamaño 6×6 y después se realizarán diversas operaciones sobre ella con el operador (:).

```
>> A=magic(6)
A =
    35     1     6    26    19    24
     3    32     7    21    23    25
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
     4    36    29    13    18    11
```

Recuérdese que MATLAB accede a los elementos de una matriz por medio de los índices de fila y de columna encerrados entre paréntesis y separados por una coma. Por ejemplo:

```
>> A(2,3)
ans =
     7
```

El siguiente comando extrae los 4 primeros elementos de la 6ª fila:

```
>> A(6, 1:4)
ans =
     4    36    29    13
```

Los dos puntos aislados representan "todos los elementos". Por ejemplo, el siguiente comando extrae todos los elementos de la 3ª fila:

```
>> A(3, :)
ans =
    31     9     2    22    27    20
```

Para acceder a la última fila o columna puede utilizarse la palabra ***end***, en lugar del número correspondiente. Por ejemplo, para extraer la sexta fila (la última) de la matriz:

```
>> A(end, :)
ans =
     4    36    29    13    18    11
```

El siguiente comando extrae todos los elementos de las filas 3, 4 y 5:

```
>> A(3:5,:)
ans =
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
```

Se pueden extraer conjuntos disjuntos de filas utilizando *corchetes* [ ]. Por ejemplo, el siguiente comando extrae las filas 1, 2 y 5:

```
>> A([1 2 5],:)
ans =
    35     1     6    26    19    24
     3    32     7    21    23    25
    30     5    34    12    14    16
```

En los ejemplos anteriores se han extraído filas y no columnas por motivos del espacio ocupado por el resultado en la hoja de papel. Es evidente que todo lo que se dice para filas vale para columnas y viceversa: basta cambiar el orden de los índices.

El operador dos puntos (:) puede utilizarse en ambos lados del operador (=). Por ejemplo, a continuación se va a definir una matriz identidad **B** de tamaño 6×6 y se van a reemplazar filas de **B** por filas de **A**. Obsérvese que la siguiente secuencia de comandos sustituye las filas 2, 4 y 5 de **B** por las filas 1, 2 y 3 de **A**,

```
>> B=eye(size(A));
>> B([2 4 5],:)=A(1:3,:)
B =
     1     0     0     0     0     0
    35     1     6    26    19    24
     0     0     1     0     0     0
     3    32     7    21    23    25
    31     9     2    22    27    20
     0     0     0     0     0     1
```

Se pueden realizar operaciones aún más complicadas, tales como la siguiente[10]:

```
>> B=eye(size(A));
>> B(1:2,:)=[0 1; 1 0]*B(1:2,:)
```

Como nuevo ejemplo, se va a ver la forma de invertir el orden de los elementos de un vector:

```
>> x=rand(1,5)
x =
    0.9103    0.7622    0.2625    0.0475    0.7361
>> x=x(5:-1:1)
x =
    0.7361    0.0475    0.2625    0.7622    0.9103
```

Obsérvese que por haber utilizado paréntesis –en vez de corchetes– los valores generados por el operador (:) afectan a los índices del vector y no al valor de sus elementos.

Para invertir el orden de las columnas de una matriz se puede hacer lo siguiente:

```
>> A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> A(:,3:-1:1)
ans =
     6     1     8
     7     5     3
     2     9     4
```

aunque hubiera sido más fácil utilizar la función ***fliplr(A)***, que es específica para ello.

Finalmente, hay que decir que ***A(:)*** representa un vector columna con las columnas de **A** una detrás de otra.

3.5.5. MATRIZ VACÍA A[ ]. BORRADO DE FILAS O COLUMNAS

Para MATLAB una matriz definida sin ningún elemento entre los corchetes es una matriz que *existe*, pero que está *vacía*, o lo que es lo mismo que tiene *dimensión cero*. Considérense los siguientes ejemplos de aplicación de las matrices vacías:

---

[10] Se sustituyen las dos primeras filas de **B** por el producto de dichas filas por una matriz de permutación.

```
>> A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> B=[]
B =
     []
>> exist(B)
ans =
     []
>> isempty(B)
ans =
     1
>> A(:,3)=[]
A =
     8     1
     3     5
     4     9
```

Las funciones ***exist()*** e ***isempty()*** permiten chequear si una variable existe y si está vacía. En el último ejemplo se ha eliminado la 3ª columna de **A** asignándole la matriz vacía.

3.5.6. DEFINICIÓN DE VECTORES Y MATRICES A PARTIR DE UN FICHERO

MATLAB acepta como entrada un fichero ***nombre.m*** (siempre con extensión ***.m***) que contiene instrucciones y/o funciones. Dicho fichero se llama desde la línea de comandos tecleando simplemente su nombre, sin la extensión. A su vez, un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso puede llamarse a sí mismo (funciones recursivas). Las variables definidas dentro de un fichero de comandos ****.m*** que se ejecuta desde la línea de comandos son variables del ***espacio de trabajo base***, esto es, pueden ser accedidas desde fuera de dicho fichero; no sucede lo mismo si el fichero ****.m*** corresponde a una función. Si un fichero de comandos se llama desde una función, las variables que se crean pertenecen al espacio de trabajo de dicha función.

Como ejemplo se puede crear un fichero llamado ***unidad.m*** que construya una matriz unidad de tamaño 3×3 llamada ***U33*** en un directorio llamado ***c:\matlab***. Este fichero deberá contener la línea siguiente:

```
U33=eye(3)
```

Desde MATLAB llámese al comando ***unidad*** y obsérvese el resultado. Entre otras razones, es muy importante utilizar ficheros de comandos para poder utilizar el ***Debugger*** y para evitar teclear muchas veces los mismos datos, sentencias o expresiones.

3.5.7. DEFINICIÓN DE VECTORES Y MATRICES MEDIANTE FUNCIONES Y DECLARACIONES

También se pueden definir las matrices y vectores por medio de ***funciones de librería*** (las que se verán en la siguiente sección) y de ***funciones programadas por el usuario*** (que también se verán más adelante).

3.6. Operadores relacionales

El lenguaje de programación de MATLAB dispone de los siguientes operadores relacionales:

<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que

==	igual que
~=	distinto que[11]

Obsérvese que, salvo el último de ellos, coinciden con los correspondientes operadores relacionales de C. Sin embargo, ésta es una coincidencia más bien formal. En MATLAB los operadores relacionales pueden aplicarse a vectores y matrices, y eso hace que tengan un significado especial.

Al igual que en C, si una comparación se cumple el resultado es 1 (***true***), mientras que si no se cumple es 0 (***false***). Recíprocamente, cualquier valor distinto de cero es considerado como ***true*** y el cero equivale a ***false***. La diferencia con C está en que cuando los operadores relacionales de MATLAB se aplican a dos matrices o vectores del mismo tamaño, ***la comparación se realiza elemento a elemento***, y ***el resultado es otra matriz de unos y ceros del mismo tamaño***, que recoge el resultado de cada comparación entre elementos. Considérese el siguiente ejemplo como ilustración de lo que se acaba de decir:

```
>> A=[1 2;0 3]; B=[4 2;1 5];
>> A==B
ans =
     0     1
     0     0
>> A~=B
ans =
     1     0
     1     1
```

3.7. Operadores lógicos

Los operadores lógicos de MATLAB son los siguientes:

&	*and* (función equivalente: **and(A,B)**). Se evalúan siempre ambos operandos, y el resultado es ***true*** sólo si ambos son ***true***.
&&	*and* breve: si el primer operando es ***false*** ya no se evalúa el segundo, pues el resultado final ya no puede ser más que ***false***.
|	*or* (función equivalente: **or(A,B)**). Se evalúan siempre ambos operandos, y el resultado es ***false*** sólo si ambos son ***false***.
||	*or* breve: si el primer operando es ***true*** ya no se evalúa el segundo, pues el resultado final no puede ser más que ***true***.
~	*negación lógica* (función equivalente: **not(A)**)
xor(A,B)	realiza un "or exclusivo", es decir, devuelve 0 en el caso en que ambos sean 1 ó ambos sean 0.

Los operadores lógicos se combinan con los relacionales para poder comprobar el cumplimiento de condiciones múltiples. Más adelante se verán otros ejemplos y ciertas funciones de las que dispone MATLAB para facilitar la aplicación de estos operadores a vectores y matrices.

Los ***operadores lógicos breves*** (&&) y (||) se utilizan para simplificar las operaciones de comparación evitando operaciones innecesarias, pero también para evitar ciertos errores que se producirían en caso de evaluar incondicionalmente el segundo argumento. Considérese por ejemplo la siguiente sentencia, que evita una división por cero:

```
r = (b~=0) && (a/b>0);
```

[11] El carácter (~) se obtiene en los PCs pulsando sucesivamente las teclas 1, 2 y 6 manteniendo ***Alt*** pulsada.

4. FUNCIONES DE LIBRERÍA

MATLAB tiene un gran número de funciones incorporadas. Algunas son ***funciones intrínsecas***, esto es, funciones incorporadas en el propio código ejecutable del programa. Estas funciones son particularmente rápidas y eficientes. Existen además funciones definidas en ficheros ***.m*** y ***.mex***[12] que vienen con el propio programa o que han sido aportadas por usuarios del mismo. Estas funciones extienden en gran manera las posibilidades del programa.

MATLAB dispone también de ficheros ***.p***, que son los ficheros ***.m*** pre-compilados con la función ***pcode***. Se verán más adelante.

Recuérdese que para que MATLAB encuentre una determinada función de usuario el correspondiente *fichero-M* debe estar en el ***directorio actual*** o en uno de los directorios del ***search path***.

4.1. Características generales de las funciones de MATLAB

El concepto de función en MATLAB es semejante al de C y al de otros lenguajes de programación, aunque con algunas diferencias importantes. Al igual que en C, una función tiene ***nombre***, ***valor de retorno*** y ***argumentos***. Una función *se llama* utilizando su nombre en una expresión o utilizándolo como un comando más. Las funciones se pueden definir en ficheros de texto ***.m*** en la forma que se verá más adelante. Considérense los siguientes ejemplos de llamada a funciones:

```
>> [maximo, posmax] = max(x);
>> r = sqrt(x^2+y^2) + eps;
>> a = cos(alfa) - sin(alfa);
```

donde se han utilizado algunas funciones matemáticas bien conocidas como el cálculo del valor máximo, el seno, el coseno y la raíz cuadrada. Los ***nombres*** de las funciones se han puesto en negrita. Los ***argumentos*** de cada función van a continuación del nombre entre paréntesis (y separados por comas si hay más de uno). Los ***valores de retorno*** son el resultado de la función y sustituyen a ésta en la expresión donde la función aparece.

Una diferencia importante con otros lenguajes es que en MATLAB las funciones pueden tener ***valores de retorno matriciales múltiples*** (ya se verá que pueden recogerse en variables *ad hoc* todos o sólo parte de estos valores de retorno), como en el primero de los ejemplos anteriores. En este caso se calcula el elemento de máximo valor en un vector, y se devuelven dos valores: el valor máximo y la posición que ocupa en el vector. Obsérvese que los 2 valores de retorno ***se recogen entre corchetes***, separados por comas.

Una característica de MATLAB es que las funciones que no tienen argumentos no llevan paréntesis, por lo que a simple vista no siempre son fáciles de distinguir de las simples variables. En la segunda línea de los ejemplos anteriores, ***eps*** es una función sin argumentos, que devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. En lo sucesivo el nombre de la función irá seguido de paréntesis si interesa resaltar que la función espera que se le pase uno o más argumentos.

Los nombres de las funciones de MATLAB ***no son palabras reservadas*** del lenguaje. Es posible crear una variable llamada ***sin*** o ***cos***, que ocultan las funciones correspondientes. Para poder acceder a las funciones hay que eliminar (***clear***) las variables del mismo nombre que las ocultan, o bien haber definido previamente una ***referencia a función*** (*function handle*). Las referencias a función se estudiarán en el apartado 6.4, a partir de la página 72.

MATLAB permite que una función tenga un ***número variable de argumentos y valores de retorno***, determinado sólo en tiempo de ejecución. Más adelante se verá cómo se hace esto.

[12] Los ficheros ***.mex*** son ficheros de código ejecutable.

MATLAB tiene diversos tipos de funciones. A continuación se enumeran los tipos de funciones más importantes, clasificadas según su finalidad:

1.- Funciones matemáticas elementales.
2.- Funciones especiales.
3.- Funciones matriciales elementales.
4.- Funciones matriciales específicas.
5.- Funciones para la descomposición y/o factorización de matrices.
6.- Funciones para análisis estadístico de datos.
7.- Funciones para análisis de polinomios.
8.- Funciones para integración de ecuaciones diferenciales ordinarias.
9.- Resolución de ecuaciones no-lineales y optimización.
10.- Integración numérica.
11.- Funciones para procesamiento de señal.

A continuación se enumeran algunas características generales de todas las funciones de MATLAB:

– Los ***argumentos actuales***[13] de estas funciones pueden ser expresiones y también llamadas a otra función.

– Las funciones de MATLAB nunca devuelven modificadas las variables que se pasan como argumentos, a no ser que se incluyan también como valores de retorno. Si el usuario las modifica dentro de la función, previamente se sacan copias de esas variables (se modifican las copias, no las variables originales). Se podría decir que los argumentos de las funciones de MATLAB siempre se pasan ***por valor***, ***nunca por referencia***.

– MATLAB admite valores de retorno matriciales múltiples. Por ejemplo, en el comando:

```
>> [V, D] = eig(A)
```

la función ***eig()*** calcula los valores y vectores propios de la matriz cuadrada **A**. Los vectores propios se devuelven como columnas de la matriz **V**, mientras que los valores propios son los elementos de la matriz diagonal **D**. En los ejemplos siguientes:

```
>> [xmax, imax] = max(x)
>> xmax = max(x)
```

puede verse que la misma función ***max()*** puede ser llamada recogiendo dos valores de retorno (el máximo elemento de un vector y la posición que ocupa) o un sólo valor de retorno (el máximo elemento).

– Las operaciones de suma y/o resta de una matriz con un escalar consisten en sumar y/o restar el escalar a todos los elementos de la matriz.

– Recuérdese que tecleando ***help nombre_funcion*** se obtiene de inmediato información sobre la función de ese nombre. En el ***Help Desk*** aparecen enlaces a "*Functions - By Cathegory*" y "*Functions – Alphabetical List*", en donde aparecen relaciones completas de las funciones disponibles en MATLAB.

4.2. Equivalencia entre comandos y funciones

Existe una equivalencia entre las funciones y los comandos con argumentos de MATLAB. Así, un comando en la forma,

```
>> comando arg1 arg2
```

[13] Los argumentos actuales son los que se utilizan en la llamada de la función

es equivalente a una función con el mismo nombre que el comando a la que los argumentos se le pasan como cadenas de caracteres,

```
>> comando('arg1', 'arg2')
```

Esta dualidad entre comandos y funciones es sobre todo útil en programación, porque permite "*construir*" los argumentos con las operaciones propias de las cadenas de caracteres.

4.3. Funciones matemáticas elementales que operan de modo escalar

Estas funciones, que comprenden las funciones matemáticas trascendentales y otras funciones básicas, cuando se aplican a una matriz actúan sobre cada elemento de la matriz como si se tratase de un escalar. Por tanto, se aplican de la misma forma a escalares, vectores y matrices. Algunas de las funciones de este grupo son las siguientes:

sin(x)	seno
cos(x)	coseno
tan(x)	tangente
asin(x)	arco seno
acos(x)	arco coseno
atan(x)	arco tangente (devuelve un ángulo entre $-\pi/2$ y $+\pi/2$)
atan2(x)	arco tangente (devuelve un ángulo entre $-\pi$ y $+\pi$); se le pasan 2 argumentos, proporcionales al seno y al coseno
sinh(x)	seno hiperbólico
cosh(x)	coseno hiperbólico
tanh(x)	tangente hiperbólica
asinh(x)	arco seno hiperbólico
acosh(x)	arco coseno hiperbólico
atanh(x)	arco tangente hiperbólica
log(x)	logaritmo natural
log10(x)	logaritmo decimal
exp(x)	función exponencial
sqrt(x)	raíz cuadrada
sign(x)	devuelve -1 si <0, 0 si $=0$ y 1 si >0. Aplicada a un número complejo, devuelve un vector unitario en la misma dirección
rem(x,y)	resto de la división (2 argumentos que no tienen que ser enteros)
mod(x,y)	similar a ***rem*** (Ver diferencias con el ***Help***)
round(x)	redondeo hacia el entero más próximo
fix(x)	redondea hacia el entero más próximo a 0
floor(x)	valor entero más próximo hacia $-\infty$
ceil(x)	valor entero más próximo hacia $+\infty$
gcd(x)	máximo común divisor
lcm(x)	mínimo común múltiplo
real(x)	partes reales
imag(x)	partes imaginarias
abs(x)	valores absolutos
angle(x)	ángulos de fase

4.4. Funciones que actúan sobre vectores

Las siguientes funciones ***sólo actúan sobre vectores*** (no sobre matrices, ni sobre escalares):

[xm,im]=max(x)	máximo elemento de un vector. Devuelve el valor máximo **xm** y la posición que ocupa **im**
min(x)	mínimo elemento de un vector. Devuelve el valor mínimo y la posición que ocupa
sum(x)	suma de los elementos de un vector
cumsum(x)	devuelve el vector suma acumulativa de los elementos de un vector (cada elemento del resultado es una suma de elementos del original)
mean(x)	valor medio de los elementos de un vector
std(x)	desviación típica
prod(x)	producto de los elementos de un vector
cumprod(x)	devuelve el vector producto acumulativo de los elementos de un vector
[y,i]=sort(x)	ordenación de menor a mayor de los elementos de un vector **x**. Devuelve el vector ordenado **y**, y un vector **i** con las posiciones iniciales en **x** de los elementos en el vector ordenado **y**.

En realidad estas funciones ***se pueden aplicar también a matrices***, pero en ese caso ***se aplican por separado a cada columna de la matriz***, dando como valor de retorno un vector resultado de aplicar la función a cada columna de la matriz considerada como vector. Si estas funciones se quieren aplicar a las filas de la matriz basta aplicar dichas funciones a la matriz traspuesta.

4.5. Funciones que actúan sobre matrices

Las siguientes funciones exigen que el/los argumento/s sean matrices. En este grupo aparecen algunas de las funciones más útiles y potentes de MATLAB. Se clasificarán en varios subgrupos:

4.5.1. FUNCIONES MATRICIALES ELEMENTALES:

B = A'	calcula la traspuesta (conjugada) de la matriz **A**
B = A.'	calcula la traspuesta (sin conjugar) de la matriz **A**
v = poly(A)	devuelve un vector **v** con los coeficientes del polinomio característico de la matriz cuadrada **A**
t = trace(A)	devuelve la traza **t** (suma de los elementos de la diagonal) de una matriz cuadrada **A**
[m,n] = size(A)	devuelve el número de filas **m** y de columnas **n** de una matriz rectangular **A**
n = size(A)	devuelve el tamaño de una matriz cuadrada **A**
nf = size(A,1)	devuelve el número de filas de **A**
nc = size(A,2)	devuelve el número de columnas de **A**

4.5.2. FUNCIONES MATRICIALES ESPECIALES

Las funciones ***exp()***, ***sqrt()*** y ***log()*** se aplican ***elemento a elemento*** a las matrices y/o vectores que se les pasan como argumentos. Existen otras funciones similares que tienen también sentido cuando se aplican a una matriz como una única entidad. Estas funciones son las siguientes (se distinguen porque llevan una "m" adicional en el nombre):

expm(A)	si **A**=**XDX'**, expm(**A**) = **X***diag(exp(diag(**D**)))***X'**
sqrtm(A)	devuelve una matriz que multiplicada por sí misma da la matriz **A**
logm()	es la función recíproca de expm(**A**)

Aunque no pertenece a esta familia de funciones, se puede considerar que el ***operador potencia*** (^) está emparentado con ellas. Así, es posible decir que:

A^n — está definida si **A** es cuadrada y **n** un número real. Si **n** es entero, el resultado se calcula por multiplicaciones sucesivas. Si **n** es real, el resultado se calcula como: **A**^n=**X*****D**.^n***X'** siendo [**X**,**D**]=eig(**A**)

4.5.3. FUNCIONES DE FACTORIZACIÓN Y/O DESCOMPOSICIÓN MATRICIAL

A su vez este grupo de funciones se puede subdividir en 4 subgrupos:

– Funciones basadas en la factorización triangular (eliminación de Gauss):

[L,U] = lu(A)	descomposición de Crout (**A** = **LU**) de una matriz. La matriz **L** es una permutación de una matriz triangular inferior (dicha permutación es consecuencia del pivotamiento por columnas utilizado en la factorización)
B = inv(A)	calcula la inversa de **A**. Equivale a **B**=inv(**U**)*inv(**L**)
d = det(A)	devuelve el determinante **d** de la matriz cuadrada **A**. Equivale a d=det(**L**)*det(**U**)
E = rref(A)	reducción a forma de escalón (mediante la eliminación de Gauss con pivotamiento por columnas, haciendo ceros también encima de los pivots) de una matriz rectangular **A**
[E,xc] = rref(A)	reducción a forma de escalón con un vector **xc** que da información sobre una posible base del espacio de columnas de **A**
U = chol(A)	descomposición de Cholesky de matriz simétrica y positivo-definida. Sólo se utiliza la diagonal y la parte triangular superior de **A**. El resultado es una matriz triangular superior tal que **A** = **U'*U**
c = rcond(A)	devuelve una estimación del recíproco de la condición numérica de la matriz **A** basada en la norma-1. Si el resultado es próximo a 1 la matriz **A** está bien condicionada; si es próximo a 0 no lo está.

– Funciones basadas en el cálculo de valores y vectores propios:

[X,D] = eig(A)	valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de una matriz cuadrada **A**. Con frecuencia el resultado es complejo (si **A** no es simétrica)
[X,D] = eig(A,B)	valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de dos matrices cuadradas **A** y **B** (**Ax** = λ**Bx**). Los vectores propios están normalizados de modo que **X'*B*X=I**. Cuando **A** es simétrica y **B** es simétrica y definida-positiva se puede utilizar [**X**,**D**] = eig(**A**,**B**,'chol').

– Funciones basadas en la descomposición QR:

[Q,R] = qr(A)	descomposición QR de una matriz rectangular. Se utiliza para sistemas con más ecuaciones que incógnitas. **Q** es una matriz ortogonal, es decir, es cuadrada aunque **A** no lo sea (*m*>*n*). No se garantiza que los elementos diagonales de **R** sean positivos, lo cual crea dificultades en algunos problemas (esa factorización no coincide con la de Gram-Schmidt).
[Q,R] = qr(A,0)	similar a la anterior, pero con **Q** del mismo tamaño que **A**, es decir, sin completar una base ortonormal cuando *m*>*n*.
[Q,R,E]=qr(A)	factorización QR con pivotamiento por columnas. La matriz **E** es una matriz de permutación tal que **A*E**=**Q*R**. La matriz **E** se determina de modo que los elementos de **abs(diag(R))** son decrecientes.
B = null(A)	devuelve una base ortonormal del subespacio nulo (kernel, o conjunto de vectores **x** tales que **Ax** = 0) de la matriz rectangular **A**, calculada mediante la descomposición de valores singulares. Las columnas de B son ortonormales: **B'*B**=**I**.

B = null(A,'r')	devuelve una base del subespaco nulo de **A** calculada a partir de la forma de escalón reducida. Las columnas de **B** no son ortonormales y se obtienen alternativamente dando valor cero a todas las variables libres excepto a una a la que se da valor unidad.
Q = orth(A)	las columnas de **Q** son una base ortonormal del espacio de columnas de **A**. El número de columnas de **Q** es el rango de **A**.

– Funciones basadas en la descomposición de valores singulares

[U,D,V] = svd(A)	descomposición de valor singular de una matriz rectangular (**A**=**U*D*V'**). **U** y **V** son matrices ortonormales. **D** es diagonal $m \times n$ (mismo tamaño que **A**) y contiene los valores singulares.
B = pinv(A)	calcula la pseudo-inversa de una matriz rectangular **A**.
r = rank(A)	calcula el rango **r** de una matriz rectangular **A**.
nor = norm(A)	calcula la norma-2 de una matriz (el mayor valor singular).
nor = norm(A,2)	lo mismo que la anterior.
nor = normest(A)	calcula de forma aproximada la norma-2 con menos operaciones aritméticas que la función ***norm***.
c = cond(A)	condición numérica sub-2 de la matriz **A**. Es el cociente entre el máximo y el mínimo valor singular. La condición numérica da una idea de los errores que se obtienen al resolver un sistema de ecuaciones lineales con dicha matriz: su logaritmo indica el número de cifras significativas que se pierden. Si **A** es grande el cálculo es laborioso.
c = condest(A)	estimación por defecto de la condición numérica de **A** con la norma-1. Esta función es mucho más económica que ***cond***.

– Cálculo del rango, normas y condición numérica:

Existen varias formas de realizar estos cálculos, con distintos niveles de esfuerzo de cálculo y de precisión en el resultado.

El rango se calcula implícitamente (sin que el usuario lo pida) al ejecutar las funciones ***rref(A)***, ***orth(A)***, ***null(A)*** y ***pinv(A)***. Con ***rref(A)*** el rango se calcula como el número de filas diferentes de cero; con ***orth(A)*** y ***null(A)*** –basadas ambas en la descomposición QR– el rango es el número de columnas del resultado (o **n** menos el número de columnas del resultado). Con ***pinv(A)*** se utiliza la descomposición de valor singular, que es el método más fiable y más caro en tiempo de *cpu*. La función ***rank(A)*** está basada en ***pinv(A)***.

Normas de matrices:

norm(A)	norma-2, es decir, máximo valor singular de **A**, ***max(svd(A))***.
normest(A)	calcula una estimación o aproximación de la norma-2. Útil para matrices grandes en las que ***norm(A)*** necesita demasiado tiempo
norm(A,2)	lo mismo que ***norm(A)***.
norm(A,1)	norma-1 de **A**, máxima suma de valores absolutos por columnas, es decir: ***max(sum(abs((A))))***.
norm(A,inf)	norma-∞ de **A**, máxima suma de valores absolutos por filas, es decir: ***max(sum(abs((A'))))***.

Normas de vectores:

norm(x,p)	norma-p, es decir ***sum(abs(x)^p)^(1/p)***.
norm(x)	norma-2 ó norma euclídea; equivale al módulo o ***norm(x,2)***.
norm(x,inf)	norma-∞, es decir ***max(abs(x))***.
norm(x,1)	norma-1, es decir ***sum(abs(x))***.

4.5.4. FUNCIÓN LINSOLVE()

La función ***linsolve*** es la forma más eficiente de que dispone MATLAB para resolver sistemas de ecuaciones lineales. A diferencia del operador barra invertida \, esta función no trata de averiguar las características de la matriz que permitan hacer una resolución más eficiente: se fía de lo que le dice el usuario. Si éste se equivoca, se obtendrá un resultado incorrecto sin ningún mensaje de error. Las formas generales de la función ***linsolve*** para resolver **Ax**=**b** son las siguientes:

```
x = linsolve(A,b)
x = linsolve(A,b,opts)
```

Obviamente, si **b** es una matriz de segundos miembros, **x** será una mariz de soluciones con el mismo nº de columnas. La primera forma de esta función utiliza la factorización LU con pivotamiento parcial si la matriz **A** es cuadrada, y la factorización QR también con pivotamiento por columnas si no lo es. La función ***linsolve*** da un ***warning*** si la matriz **A** es cuadrada y está mlas condicionada, o si es rectangular y de rango deficiente. Estos warnings se suprimen si se recoge un segundo valor de retorno **r**, que representa el inverso de la condición numérica si **A** es cuadrada o el rango si no lo es:

```
[x,r] = linsolve(A,b)
```

El argumento opcional ***opts*** representa una estructura por medio de la cual el programador proporciona información sobre las características de la matriz.. Los campos de esta estructura se pueden poner a ***true*** o a ***false***, y son los siguientes: LT (triangular inferior), UT (triangular superior), UHESS (forma de Hessenberg superior), SYM (simétrica), POSDEF (definida positiva), RECT (rectangular general) y TRANSA (se desea resolver $\mathbf{A}^T\mathbf{x}=\mathbf{b}$, en lugar de $\mathbf{Ax}=\mathbf{b}$). Obviamente, no todas estas características son compatibles entre sí; las que lo son se indican en la Tabla siguiente:

Tabla 1. Posibles campos de la esructura ***opts*** compatibles entre sí.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
true	false	false	false	false	true/false	true/false
false	true	false	false	false	true/false	true/false
false	false	true	false	false	false	true/false
false	false	false	true	true	false	true/false
false	false	false	false	false	true/false	true/false

En la Tabla 1 se observa que, en la actual versión de MATLAB, sólo se admiten matrices simétricas que son al mismo tiempo definidas positivas. Para concluir este apartado, considérense los ejemplos siguientes:

```
>> opts.LT=true;  x=linsolve(L,b,opts);
>> clear opts;  opts.SYM=true; opts.POSDEF=true;  x=linsolve(A,b,opts);
```

Obsérvese que, antes de realizar una nueva ejecución se han borrado las opciones utilizadas en la ejecución anterior.

4.6. Más sobre operadores relacionales con vectores y matrices

Cuando alguno de los operadores relacionales vistos previamente (<, >, <=, >=, == y ~=) actúa entre dos matrices (vectores) del mismo tamaño, el resultado es otra matriz (vector) de ese mismo tamaño conteniendo unos y ceros, según los resultados de cada comparación entre elementos hayan sido ***true*** o ***false***, respectivamente.

Por ejemplo, supóngase que se define una matriz *magic* **A** de tamaño 3x3 y a continuación se forma una matriz binaria **M** basada en la condición de que los elementos de **A** sean mayores que 4 (MA-

TLAB convierte este cuatro en una matriz de cuatros de modo automático). Obsérvese con atención el resultado:

```
>> A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> M=A>4
M =
     1     0     1
     0     1     1
     0     1     0
```

De ordinario, las matrices "binarias" que se obtienen de la aplicación de los operadores relacionales no se almacenan en memoria ni se asignan a variables, sino que se procesan sobre la marcha. MATLAB dispone de varias funciones para ello. Recuérdese que cualquier valor distinto de cero equivale a ***true***, mientras que un valor cero equivale a ***false***. Algunas de estas funciones son:

any(x)	función vectorial; chequea si *alguno* de los elementos del vector **x** cumple una determinada condición (en este caso ser distinto de cero). Devuelve un uno ó un cero
any(A)	se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros
all(x)	función vectorial; chequea si *todos* los elementos del vector **x** cumplen una condición. Devuelve un uno ó un cero
all(A)	se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros
find(x)	busca índices correspondientes a elementos de vectores que cumplen una determinada condición. El resultado es un vector con los índices de los elementos que cumplen la condición
find(A)	cuando esta función se aplica a una matriz la considera como un vector con una columna detrás de otra, de la 1ª a la última.

A continuación se verán algunos ejemplos de utilización de estas funciones.

```
>> A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> m=find(A>4)
m =
     1
     5
     6
     7
     8
```

Ahora se van a sustituir los elementos que cumplen la condición anterior por valores de 10. Obsérvese cómo se hace y qué resultado se obtiene:

```
>> A(m)=10*ones(size(m))
A =
    10     1    10
     3    10    10
     4    10     2
```

donde ha sido necesario convertir el 10 en un vector del mismo tamaño que **m**. Para chequear si hay algún elemento de un determinado valor –por ejemplo 3– puede hacerse lo siguiente:

```
>> any(A==3)
ans =
     1     0     0
>> any(ans)
ans =
     1
```

mientras que para comprobar que todos los elementos de **A** son mayores que cero:

```
>> all(all(A))
ans =
     1
```

En este caso no ha hecho falta utilizar el operador relacional porque cualquier elemento distinto de cero equivale a ***true***.

La función ***isequal(A, B)*** devuelve *uno* si las matrices son idénticas y *cero* si no lo son.

4.7. Otras funciones que actúan sobre vectores y matrices

Las siguientes funciones pueden actuar sobre vectores y matrices, y sirven para chequear ciertas condiciones:

exist('var')	comprueba si el nombre **var** existe como variable, función, directorio, fichero, etc.
isnan(A)	chequea si hay valores ***NaN*** en **A**, devolviendo una matriz de unos y ceros del mismo tamaño que **A**.
isinf(A)	chequea si hay valores ***Inf*** en **A**, devolviendo una matriz de unos y ceros del mismo tamaño que **A**.
isfinite(A)	chequea si los valores de **A** son finitos.
isempty(A)	chequea si un vector o matriz está vacío o tiene tamaño nulo.
ischar()	chequea si una variable es una cadena de caracteres (*string*).
isglobal()	chequea si una variable es global.
issparse()	chequea si una matriz es dispersa (*sparse*, es decir, con un gran número de elementos cero).

A continuación se presentan algunos ejemplos de uso de estas funciones en combinación con otras vistas previamente. Se define un vector **x** con un ***NaN***, que se elimina en la forma:

```
>> x=[1 2 3 4 0/0 6]
Warning: Divide by zero
x =
     1     2     3     4   NaN     6
>> i=find(isnan(x))
i =
     5
>> x=x(find(~isnan(x)))
x =
     1     2     3     4     6
```

Otras posibles formas de eliminarlo serían las siguientes:

```
>> x=x(~isnan(x))
>> x(isnan(x))=[]
```

La siguiente sentencia elimina las filas de una matriz que contienen algún ***NaN***:

```
>> A(any(isnan(A)'), :)=[]
```

4.8. Determinación de la fecha y la hora

MATLAB dispone de funciones que dan información sobre la ***fecha*** y la ***hora*** actual (la del reloj del ordenador). Las funciones más importantes relacionadas con la fecha y la hora son las siguientes.

clock	devuelve un vector fila de seis elementos que representan el ***año***, el ***mes***, el ***día***, la ***hora***, los ***minutos*** y los ***segundos***, según el reloj interno del computador. Los cinco primeros son valores enteros, pero la cifra correspondiente a los segundos contiene información hasta las milésimas de segundo.
now	devuelve un número (***serial date number***) que contiene toda la información de la fecha y hora actual. Se utiliza como argumento de otras funciones.
date	devuelve la fecha actual como cadena de caracteres (por ejemplo: *24-Aug-2004*).
datestr(t)	convierte el *serial date number* ***t*** en cadena de caracteres con el *día*, *mes*, *año*, *hora*, *minutos* y *segundos*. Ver en los manuales on-line los formatos de cadena admitidos.
datenum()	convierte una cadena ('mes-día-año') o un conjunto de seis números (año, mes, día, horas, minutos, segundos) en *serial date number*.
datevec()	convierte *serial date numbers* o cadenas de caracteres en el vector de seis elementos que representa la fecha y la hora.
calendar()	devuelve una matriz 6×7 con el calendario del mes actual, o del mes y año que se especifique como argumento.
weekday(t)	devuelve el día de la semana para un *serial date number* ***t***.

4.9. Funciones para cálculos con polinomios

Para MATLAB un polinomio se puede definir mediante un vector de coeficientes. Por ejemplo, el polinomio:

$$x^4 - 8x^2 + 6x - 10 = 0$$

se puede representar mediante el vector [1, 0, -8, 6, -10]. MATLAB puede realizar diversas operaciones sobre él, como por ejemplo evaluarlo para un determinado valor de **x** (función ***polyval()***) y calcular las raíces (función ***roots()***):

```
>> pol=[1 0 -8 6 -10]
pol =
     1     0    -8     6   -10
>> roots(pol)
ans =
  -3.2800
   2.6748
   0.3026 + 1.0238i
   0.3026 - 1.0238i
>> polyval(pol,1)
ans =
   -11
```

Para calcular producto de polinomios MATLAB utiliza una función llamada ***conv()*** (de *producto de convolución*). En el siguiente ejemplo se va a ver cómo se multiplica un polinomio de segundo grado por otro de tercer grado:

```
>> pol1=[1 -2 4]
pol1 =
     1    -2     4
```

```
>> pol2=[1 0 3 -4]
pol2 =
     1     0     3    -4
>> pol3=conv(pol1,pol2)
pol3 =
     1    -2     7   -10    20   -16
```

Para dividir polinomios existe otra función llamada ***deconv()***. Las funciones orientadas al cálculo con polinomios son las siguientes:

poly(A)	polinomio característico de la matriz **A**
roots(pol)	raíces del polinomio **pol**
polyval(pol,x)	evaluación del polinomio **pol** para el valor de **x**. Si **x** es un vector, **pol** se evalúa para cada elemento de **x**
polyvalm(pol,A)	evaluación del polinomio **pol** de la matriz **A**
conv(p1,p2)	producto de convolución de dos polinomios **p1** y **p2**
[c,r]=deconv(p,q)	división del polinomio **p** por el polinomio **q**. En **c** se devuelve el cociente y en **r** el resto de la división
residue(p1,p2)	descompone el cociente entre **p1** y **p2** en suma de fracciones simples (ver >>***help residue***)
polyder(pol)	calcula la derivada de un polinomio
polyder(p1,p2)	calcula la derivada de producto de polinomios
polyfit(x,y,n)	calcula los coeficientes de un polinomio **p(x)** de grado **n** que se ajusta a los datos **p(x(i)) ~= y(i)**, en el sentido de mínimo error cuadrático medio.
interp1(xp,yp,x)	calcula el valor interpolado para la abscisa **x** a partir de un conjunto de puntos dado por los vectores **xp** e **yp**.
interp1(xp,yp,x,'m')	como la anterior, pero permitiendo especificar también el método de interpolación. La cadena de caracteres **m** admite los valores 'nearest', 'linear', 'spline', 'pchip', 'cubic' y 'v5cubic'.

5. OTROS TIPOS DE DATOS DE MATLAB

En los capítulos precedentes se ha visto la "especialidad" de MATLAB: trabajar con vectores y matrices. En este capítulo se va a ver que MATLAB puede también trabajar con otros tipos de datos:

1. Conjuntos o cadenas de caracteres, fundamentales en cualquier lenguaje de programación.
2. Hipermatrices, o matrices de más de dos dimensiones.
3. Estructuras, o agrupaciones bajo un mismo nombre de datos de naturaleza diferente.
4. Vectores o matrices de celdas (cell arrays), que son vectores o matrices cuyos elementos pueden ser cualquier otro tipo de dato.
5. Matrices dispersas o matrices dispersas, que son matrices que pueden ser de muy gran tamaño con la mayor parte de sus elementos cero.

5.1. Cadenas de caracteres

MATLAB trabaja también con ***cadenas de caracteres***, con ciertas semejanzas y también diferencias respecto a C/C++ y Java. A continuación se explica lo más importante del manejo de cadenas de caracteres en MATLAB. Las funciones para cadenas de caracteres están en el sub-directorio ***toolbox\matlab\strfun*** del directorio en que esté instalado MATLAB.

Los caracteres de una cadena se almacenan en un vector, con un carácter por elemento. Cada carácter ocupa dos bytes. Las cadenas de caracteres van entre ***apóstrofos*** o ***comillas simples***, como por ejemplo: 'cadena'. Si la cadena debe contener comillas, éstas se representan por un doble carácter comilla, de modo que se pueden distinguir fácilmente del principio y final de la cadena. Por ejemplo, para escribir la cadena **ni 'idea'** se escribiría **'ni''idea'''**.

Una ***matriz de caracteres*** es una matriz cuyos elementos son caracteres, o bien una matriz cuyas filas son cadenas de caracteres. Todas las filas de una ***matriz de caracteres*** deben tener el ***mismo número de elementos***. Si es preciso, las cadenas (filas) más cortas se completan con blancos.

A continuación se pueden ver algunos ejemplos y practicar con ellos:

```
>> c='cadena'
c =
cadena
>> size(c)              % dimensiones del array
ans =
     1     6
>> double(c)            % convierte en números ASCII cada carácter
ans =
    99    97   100   101   110    97
>> char(abs(c))         % convierte números ASCII en caracteres
ans =
cadena
>> cc=char('más','madera')   % convierte dos cadenas en una matriz
cc =
más
madera
>> size(cc)             % se han añadido tres espacios a 'más'
ans =
     2     6
```

Las funciones más importantes para manejo de cadenas de caracteres son las siguientes:

double(c)	convierte en números ASCII cada carácter
char(v)	convierte un vector de números **v** en una cadena de caracteres

char(c1,c2)	crea una matriz de caracteres, completando con blancos las cadenas más cortas
deblank(c)	elimina los blancos al final de una cadena de caracteres
disp(c)	imprime el texto contenido en la variable **c**
ischar(c)	detecta si una variable es una cadena de caracteres
isletter()	detecta si un carácter es una letra del alfabeto. Si se le pasa un vector o matriz de caracteres devuelve un vector o matriz de unos y ceros
isspace()	detecta si un carácter es un espacio en blanco. Si se le pasa un vector o matriz de caracteres devuelve un vector o matriz de unos y ceros
strcmp(c1,c2)	comparación de cadenas. Si las cadenas son iguales devuelve un uno, y si no lo son, devuelve un cero (funciona de modo diferente que la correspondiente función de C)
strcmpi(c1,c2)	igual que ***strcmp(c1,c2)***, pero ignorando la diferencia entre mayúsculas y minúsculas
strncmp(c1,c2,n)	compara los **n** primeros caracteres de dos cadenas
c1==c2	compara dos cadenas carácter a carácter. Devuelve un vector o matriz de unos y ceros
s=[s,' y más']	concatena cadenas, añadiendo la segunda a continuación de la primera
findstr(c1,c2)	devuelve un vector con las posiciones iniciales de todas las veces en que la cadena más corta aparece en la más larga
strmatch(cc,c)	devuelve los índices de todos los elementos de la matriz de caracteres (o vector de celdas) **cc**, que empiezan por la cadena **c**
strrep(c1,c2,c3)	sustituye la cadena **c2** por **c3**, cada vez que **c2** es encontrada en **c1**
[p,r]=strtok(t)	separa las palabras de una cadena de caracteres **t**. Devuelve la primera palabra **p** y el resto de la cadena **r**
int2str(v)	convierte un número entero en cadena de caracteres
num2str(x,n)	convierte un número real **x** en su expresión por medio de una cadena de caracteres, con cuatro cifras decimales por defecto (pueden especificarse más cifras, con un argumento opcional **n**)
str2double(str)	convierte una cadena de caracteres representando un número real en el número real correspondiente
vc=cellstr(cc)	convierte una matriz de caracteres **cc** en un vector de celdas **vc**, eliminando los blancos adicionales al final de cada cadena. La función ***char()*** realiza las conversiones opuestas
sprintf	convierte valores numéricos en cadenas de caracteres, de acuerdo con las reglas y formatos de conversión del lenguaje C. Esta es la función más general para este tipo de conversión y se verá con mas detalle en la Sección 6.6.2.

Con las funciones anteriores se dispone en MATLAB de una amplia gama de posibilidades para trabajar con cadenas de caracteres.

A continuación se pueden ver algunos ejemplos:

```
>> num2str(pi)   % el resultado es una cadena de caracteres, no un número
ans =
3.142
>> num2str(pi,8)
ans =
3.1415927
```

Es habitual convertir los valores numéricos en cadenas de caracteres para poder imprimirlos como títulos en los dibujos o gráficos. Véase el siguiente ejemplo:

```
>> fahr=70; grd=(fahr-32)/1.8;
>> title(['Temperatura ambiente: ',num2str(grd),' grados centígrados'])
```

5.2. Hipermatrices (arrays de más de dos dimensiones)

MATLAB permite trabajar con ***hipermatrices***, es decir con matrices de más de dos dimensiones (Figura 27). Una posible aplicación es almacenar con un único nombre distintas matrices del mismo tamaño (resulta una hipermatriz de 3 dimensiones). Los elementos de una hipermatriz pueden ser números, caracteres, estructuras, y vectores o matrices de celdas.

El tercer subíndice representa la tercera dimensión: la "profundidad" de la hipermatriz.

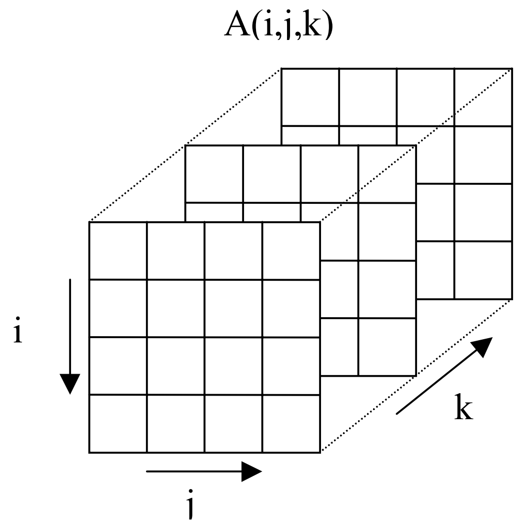


Figura 27. Hipermatriz de tres dimensiones.

5.2.1. Definición de hipermatrices

Las funciones para trabajar con estas hipermatrices están en el sub-directorio ***toolbox\matlab\datatypes***. Las funciones que operan con matrices de más de dos dimensiones son análogas a las funciones vistas previamente, aunque con algunas diferencias. Por ejemplo, las siguientes sentencias generan, en dos pasos, una matriz de 2×3×2:

```
>> AA(:,:,1)=[1 2 3; 4 5 6]   % matriz inicial
AA =
     1     2     3
     4     5     6
>> AA(:,:,2)=[2 3 4; 5 6 7]   % se añade una segunda matriz
AA(:,:,1) =
     1     2     3
     4     5     6
AA(:,:,2) =
     2     3     4
     5     6     7
```

5.2.2. Funciones que trabajan con hipermatrices

Algunas funciones de MATLAB para generar matrices admiten más de dos subíndices y pueden ser utilizadas para generar hipermatrices. Entre ellas están ***rand()***, ***randn()***, ***zeros()*** y ***ones()***. Por ejemplo, véase la siguiente sentencia y su resultado:

```
>> BB=randn(2,3,2)
BB(:,:,1) =
   -0.4326    0.1253   -1.1465
   -1.6656    0.2877    1.1909
BB(:,:,2) =
    1.1892    0.3273   -0.1867
   -0.0376    0.1746    0.7258
```

La función ***cat()*** permite concatenar matrices según las distintas "dimensiones", como puede verse en el siguiente ejemplo:

```
>> A=zeros(2,3); B=ones(2,3);
```

```
>> cat(1,A,B)
ans =
     0     0     0
     0     0     0
     1     1     1
     1     1     1
>> cat(2,A,B)
ans =
     0     0     0     1     1     1
     0     0     0     1     1     1
>> cat(3,A,B)
ans(:,:,1) =
     0     0     0
     0     0     0
ans(:,:,2) =
     1     1     1
     1     1     1
```

Las siguientes funciones de MATLAB se pueden emplear también con hipermatrices:

size()	devuelve tres o más valores (el nº de elementos en cada dimensión)
ndims()	devuelve el número de dimensiones
squeeze()	elimina las dimensiones que son igual a uno
reshape()	distribuye el mismo número de elementos en una matriz con distinta forma o con distintas dimensiones
permute(A,v)	permuta las dimensiones de **A** según los índices del vector **v**
ipermute(A,v)	realiza la permutación inversa

Respecto al resto de las funciones de MATLAB, se pueden establecer las siguientes reglas para su aplicación a hipermatrices:

1. Todas las funciones de MATLAB que operan sobre escalares (***sin()***, ***cos()***, etc.) se aplican sobre hipermatrices elemento a elemento (igual que sobre vectores y matrices). Las operaciones con escalares también se aplican de la misma manera.
2. Las funciones que operan sobre vectores (***sum()***, ***max()***, etc.) se aplican a matrices e hipermatrices según la primera dimensión, resultando un array de una dimensión inferior.
3. Las funciones matriciales propias del Álgebra Lineal (***det()***, ***inv()***, etc.) no se pueden aplicar a hipermatrices. Para poderlas aplicar hay que extraer primero las matrices correspondientes (por ejemplo, con el operador dos puntos (:)).

5.3. Estructuras

Una estructura (***struct***) es una agrupación de datos de tipo diferente bajo un mismo nombre. Estos datos se llaman ***miembros*** (*members*) o ***campos*** (*fields*). Una estructura es un nuevo tipo de dato, del que luego se pueden crear muchas variables (***objetos*** o *instances*). Por ejemplo, la estructura ***alumno*** puede contener los campos ***nombre*** (una cadena de caracteres) y ***carnet*** (un número).

5.3.1. CREACIÓN DE ESTRUCTURAS

En MATLAB la estructura ***alumno*** se crea creando un objeto de dicha estructura. A diferencia de otros lenguajes de programación, no hace falta definir previamente el modelo o patrón de la estructura. Una posible forma de hacerlo es crear uno a uno los distintos campos, como en el ejemplo siguiente:

```
>> alu.nombre='Miguel'
alu =
    nombre: 'Miguel'
```

```
>> alu.carnet=75482
alu =
    nombre: 'Miguel'
    carnet: 75482
>> alu
alu =
    nombre: 'Miguel'
    carnet: 75482
```

Se accede a los miembros o campos de una estructura por medio del ***operador punto*** (.), que une el nombre de la estructura y el nombre del campo (por ejemplo: ***alu.nombre***).

También puede crearse la estructura por medio de la función ***struct()***, como por ejemplo,

```
>> al = struct('nombre', 'Ignacio', 'carnet', 76589)
al =
    nombre: 'Ignacio'
    carnet: 76589
```

Los ***nombres de los campos*** se pasan a la función ***struct()*** entre apóstrofos ('), seguidos del valor que se les quiere dar. Este valor puede ser la cadena vacía ('') o la matriz vacía ([]).

Pueden crearse vectores y matrices (e hipermatrices) de estructuras. Por ejemplo, la sentencia,

```
>> alum(10) = struct('nombre', 'Ignacio', 'carnet', 76589)
```

crea un vector de 10 elementos cada uno de los cuales es una estructura tipo ***alumno***. Sólo el elemento 10 del vector es inicializado con los argumentos de la f***unción struct()***; el resto de los campos se inicializan con una cadena vacía o una matriz vacía[14]. Para dar valor a los campos de los elementos restantes se puede utilizar un bucle ***for*** con sentencias del tipo:

```
>> alum(i).nombre='Noelia', alum(i).carnet=77524;
```

MATLAB permite añadir un nuevo campo a una estructura en cualquier momento. La siguiente sentencia añade el campo ***edad*** a todos los elementos del vector ***alum***, aunque sólo se da valor al campo del elemento 5,

```
>> alum(5).edad=18;
```

Para ver el campo ***edad*** en los 10 elementos del vector puede teclearse el comando:

```
>> alum.edad
```

5.3.2. FUNCIONES PARA OPERAR CON ESTRUCTURAS

Las estructuras de MATLAB disponen de funciones que facilitan su uso. Algunas de estas funciones son las siguientes:

fieldnames()	devuelve un vector de celdas con cadenas de caracteres que recogen los nombres de los campos de una estructura
isfield(ST,s)	permite saber si la cadena **s** es un campo de una estructura **ST**
isstruct(ST)	permite saber si **ST** es o no una estructura
rmfield(ST,s)	elimina el campo **s** de la estructura **ST**
getfield(ST,s)	devuelve el valor del campo especificado. Si la estructura es un array hay que pasarle los índices como *cell array* (entre llaves {}) como segundo argumento
setfield(ST,s,v)	da el valor **v** al campo **s** de la estructura **ST**. Si la estructura es un array, hay que pasarle los índices como *cell array* (entre llaves {}) como segundo argumento

[14] Esta forma de crear arrays de estructuras da error si la estructura ha sido previamente declarada ***global***.

MATLAB permite definir ***estructuras anidadas***, es decir una estructura con campos que sean otras estructuras. Para acceder a los campos de la estructura más interna se utiliza dos veces el operador punto (.), como puede verse en el siguiente ejemplo, en el que la estructura ***clase*** contiene un campo que es un vector ***alum*** de alumnos,

```
>> clase=struct('curso','primero','grupo','A', ...
               'alum', struct('nombre','Juan', 'edad', 19))
clase =
    curso: 'primero'
    grupo: 'A'
     alum: [1x1 struct]
>> clase.alum(2).nombre='María';
>> clase.alum(2).edad=17;
>> clase.alum(2)
ans =
    nombre: 'María'
      edad: 17
>> clase.alum(1)
ans =
    nombre: 'Juan'
      edad: 19
```

Las estructuras se generalizan con las ***clases*** y los ***objetos***, que no se verán en este manual.

5.4. Vectores o matrices de celdas (*Cell Arrays*)

Un vector (matriz o hipermatriz) de celdas es un vector (matriz o hipermatriz) cuyos elementos son cada uno de ellos una variable de tipo cualquiera. En un array ordinario todos sus elementos son números o cadenas de caracteres. Sin embargo, en un ***array de celdas***, el primer elemento puede ser un número; el segundo una matriz; el tercero una cadena de caracteres; el cuarto una estructura, etc.

5.4.1. CREACIÓN DE VECTORES Y MATRICES DE CELDAS

Obsérvese por ejemplo cómo se crea, utilizando ***llaves*** {}, el siguiente vector de celdas,

```
>> vc(1)={[1 2 3]}
vc =
    [1x3 double]
>> vc(2)={'mi nombre'}
vc =
    [1x3 double]    'mi nombre'
>> vc(3)={rand(3,3)}
vc =
    [1x3 double]    'mi nombre'    [3x3 double]
```

Es importante que el nombre del vector de celdas ***vc*** no haya sido utilizado previamente para otra variable (si así fuera, se obtendría un error). Si es preciso se utiliza el comando ***clear***.

Obsérvese que para crear un vector de celdas los valores asignados a cada elemento se han definido entre ***llaves*** {...}.

Otra nomenclatura alternativa y similar, que también utiliza llaves, es la que se muestra a continuación:

```
>> vb{1}=[1 2 3]
vb =
    [1x3 double]
>> vb{2}='mi nombre'
vb =
    [1x3 double]    'mi nombre'
```

```
>> vb{3}=rand(3,3)
vb =
    [1x3 double]    'mi nombre'    [3x3 double]
```

y también es posible crear el vector de celdas en una sola operación en la forma,

```
vcc = {[1 2 3], 'mi nombre', rand(3,3)}
vcc =
    [1x3 double]    'mi nombre'    [3x3 double]
```

5.4.2. FUNCIONES PARA TRABAJAR CON VECTORES Y MATRICES DE CELDAS

MATLAB dispone de las siguientes funciones para trabajar con *cell arrays*:

cell(m,n)	crea un *cell array* vacío de **m** filas y **n** columnas
celldisp(ca)	muestra el contenido de todas las celdas de **ca**
cellplot(ca)	muestra una representación gráfica de las distintas celdas
iscell(ca)	indica si **ca** es un vector de celdas
num2cell()	convierte un array numérico en un *cell array*
cell2struct()	convierte un *cell array* en una estructura (ver Sección 5.4.3)
struct2cell()	convierte una estructura en un *cell array* (ver Sección 5.4.3)

5.4.3. CONVERSIÓN ENTRE ESTRUCTURAS Y VECTORES DE CELDAS

El siguiente ejemplo convierte el *cell array* ***vcc*** creado previamente en una estructura ***ST*** cuyos ***campos*** se pasan como argumentos a la función ***cell2struct()***. El tercer argumento (un 2) indica que es la segunda dimensión del *cell array* (las columnas) la que va a dar origen a los campos de la estructura. Con posterioridad la estructura ***ST*** se convierte en un nuevo *cell array* llamado ***vbb***,

```
>> ST=cell2struct(vb,{'vector','cadena','matriz'},2)
ST =
    vector: [1 2 3]
    cadena: 'mi nombre'
    matriz: [3x3 double]
>> vbb = struct2cell(ST)'    % hay que transponer para obtener una fila
vbb =
    [1x3 double]    'mi nombre'    [3x3 double]
```

La gran ventaja de las estructuras y los arrays de celdas es que proporcionan una gran flexibilidad para el almacenamiento de los más diversos tipos de información. El inconveniente es que se pierde parte de la eficiencia que MATLAB tiene trabajando con vectores y matrices.

5.5. Matrices dispersas (sparse)

Las matrices dispersas o sparse son matrices de un gran tamaño con la mayor parte de sus elementos cero. Operar sobre este tipo de matrices con los métodos convencionales lleva a obtener tiempos de cálculo prohibitivos. Por esta razón se han desarrollado técnicas especiales para este tipo de matrices. En ingeniería es muy frecuente encontrar aplicaciones en las que aparecen matrices sparse. MATLAB dispone de numerosas funciones para trabajar con estas matrices.

Las matrices dispersas se almacenan de una forma especial: solamente se guardan en memoria los elementos distintos de cero, junto con la posición que ocupan en la matriz. MATLAB usa 3 arrays para matrices reales sparse con ***nnz*** elementos distintos de cero:

1. Un array con todos los elementos distintos de cero (*nnz* elementos)
2. Un array con los índices de fila de los elementos distintos de cero (*nnz* elementos)
3. Un array con punteros a la posición del primer elemento de cada columna (*n* elementos)

En total se requiere una memoria de (*nnz**8+(*nnz*+*n*)*4) bytes. La Figura 28 muestra un ejemplo de matriz dispersa que viene con MATLAB (se puede cargar con ***load west0479***). Esta matriz tiene 479 filas y columnas. De los 229441 elementos sólo 1887 son distintos de cero. Se comprende que se pueden conseguir grandes ahorros de memoria y de tiempo de cálculo almacenando y operando sólo con los elementos distintos de cero.

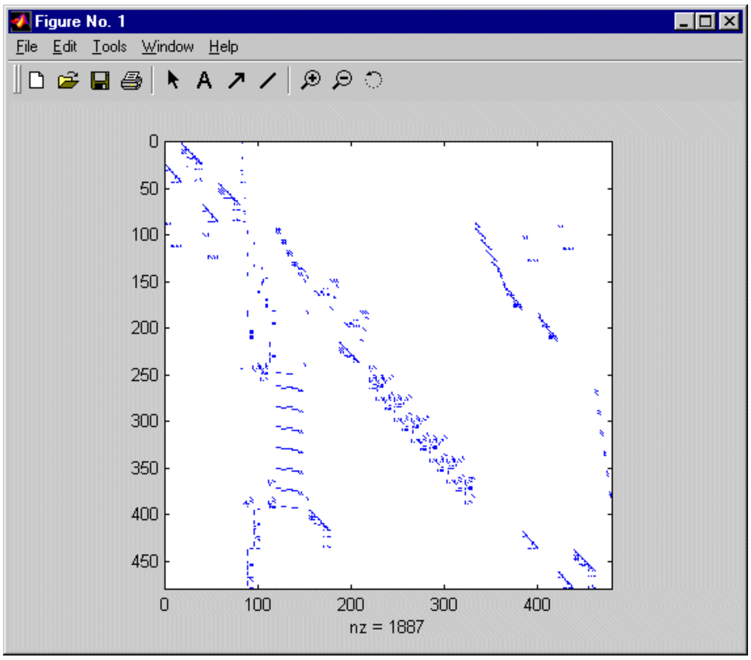


Figura 28. Ejemplo de matriz dispersa (west0479).

A continuación se va a mostrar con un ejemplo más pequeño cómo guarda MATLAB estas matrices. Primero se creará una matriz 5×5 llena y luego se convertirá en dispersa.

```
>> A=[1, 0, 0, -1, 0;  0, 2, 0, 0, 1;  0, 0, 1, 1, 0; 0, 2, 0, 1, 0;  -3, 0, 0, 0, 2]
A =
     1     0     0    -1     0
     0     2     0     0     1
     0     0     1     1     0
     0     2     0     1     0
    -3     0     0     0     2
>> S=sparse(A)
S =
   (1,1)        1
   (5,1)       -3
   (2,2)        2
   (4,2)        2
   (3,3)        1
   (1,4)       -1
   (3,4)        1
   (4,4)        1
   (2,5)        1
   (5,5)        2
```

Puede observarse cómo MATLAB muestra las matrices dispersas: primero los dos índices, de filas y de columnas, entre paréntesis y después el valor del elemento. Los elementos se almacenan por columnas y por eso se guarda la posición en que empieza cada columna.

5.5.1. FUNCIONES PARA CREAR MATRICES DISPERSAS (DIRECTORIO SPARFUN)

Las siguientes funciones permiten crear matrices dispersas. Casi todas estas funciones tienen muchas posibles formas de ser utilizadas, con distintos argumentos y valores de retorno. Se recomienda ver el ***Help*** de MATLAB para tener una información más detallada.

speye(m,n)	Matriz identidad dispersa de tamaño *m*×*n* con unos en la diagonal
sprand(m,n)	Matriz aleatoria dispersa con distribución uniforme

sprandn(m,n)	Matriz aleatoria dispersa con distribución normal
sprandsym(n)	Matriz aleatoria simétrica
spdiags(A)	Matriz dispersa a partir de las diagonales de otra matriz
sparse(m,n)	Crea una matriz dispersa de tamaño *m*×*n* con todos los elementos cero
sparse(A)	Crea una matriz dispersa a partir de una matriz llena
sparse(i,j,val,m,n)	Construye una matriz dispersa a partir de: **i** vector de índices de fila, **j** vector de índices de columna, **val** vector de valores, **m** número de filas, **n** número de columnas, y un 6º argumento que permite definir el máximo **nnz** (por defecto en el tamaño de **val**) por si se quieren añadir después más elementos
full(S)	Convierte una matriz dispersa en una matriz llena
find(S)	Encuentra los índices de los elementos distintos de cero y los devuelve como si la matriz fuera un vector (por columnas).
[i,j,val]=find(S)	Devuelve índices de fila, de columna y valores de los elementos, a partir de los cuáles se puede volver a crear la matriz
spy(S)	Representa en una figura los elementos distintos de cero de la matriz
nnz(S)	Devuelve el número de elementos distintos de cero
nonzeros(S)	Devuelve un vector lleno que contiene los elementos distintos de cero
nzmax(S)	Memoria reservada para elementos distintos de cero
spones(S)	Reemplazar los elementos distintos de cero por unos
spalloc(m,n,nzmax)	Reserva espacio para una matriz dispersa *m*×*n*
issparse(S)	Devuelve *true* si el argumento es una matriz dispersa

5.5.2. OPERACIONES CON MATRICES DISPERSAS

Las matrices dispersas son más "delicadas" que las matrices llenas. En concreto, son muy sensibles a la ordenación de sus filas y columnas. El problema no es tanto la matriz dispersa en sí, como las matrices –también dispersas- que resultan de las factorizaciones LU o de Cholesky necesarias para resolver sistemas de ecuaciones, calcular valores y vectores propios, etc. En estas factorizaciones puede haber muchos elementos cero que dejan de serlo y esto es un grave problema para la eficiencia de los cálculos. Reordenando las filas y columnas de una matriz dispersa se puede minimizar el número de elementos que se hacen distintos de cero al factorizar (*llenado* o *fill-in*). MATLAB dispone de dos formas principales de reordenación: los métodos del mínimo grado (*minimum degree*) y de Cuthill-McKee inverso (*reversed Cuthill-McKee*). A continuación se describen las funciones más importantes de MATLAB en esta categoría.

spfun('fun', S)	Aplica una función a los elementos distintos de cero de la matriz **S**
p=colmmd(S)	Devuelve el vector de permutaciones de columnas calculado con el método del mínimo grado (minimum degree). Para matrices no simétricas esta permutación tiende a producir factorizaciones LU más dispersas.
p=symmmd(S)	Devuelve el vector de permutaciones de filas y columnas (*symmetric minimum degree permutation*). Aplicando esta permutación a las filas y columnas se obtienen factorizaciones de Cholesky más dispersas.
p=symrcm(S)	Obtiene un vector de permutaciones por el método de Cuthill-McKee inverso tal que, aplicado a filas y columnas de **S**, obtiene matrices con los elementos agrupados alrededor de la diagonal principal (mínima anchura de banda). Se aplica a matrices simétricas y no simétricas.

p=colperm(S)	Obtiene una permutación de columnas que ordena las columnas en orden de número de ceros no decreciente. A veces se utiliza para ordenar antes de aplicar la factorización LU. Si la matriz es simétrica la permutación se puede aplicar a filas y a columnas.
randperm(n)	Calcula una permutación aleatoria de los *n* primeros números naturales

5.5.3. OPERACIONES DE ÁLGEBRA LINEAL CON MATRICES DISPERSAS

A continuación se describen muy brevemente las funciones de MATLAB que pueden utilizarse para operar con matrices dispersas. Algunas de estas funciones se llaman igual que las correspondientes funciones para matrices llenas, y otras son específicas de matrices dispersas. Casi todas estas funciones tienen varias formas de utilizarse. Para más detalles se sugiere recurrir al ***Help***.

[L,U,P]=lu(S)	Realiza la factorización LU
L=chol(S)	Realiza la factorización de Cholesky
[Q,R]=qr(S)	Realiza la factorización QR
[L,U]=luinc(A,tol)	Realiza una factorización LU incompleta
L=cholinc(S)	Calcula una factorización de Cholesky incompleta (Ver la Ayuda)

[V,D,FLAG] = eigs(S)	Calcula algunos valores propios de una matriz cuadrada. Esta función tiene muchas posibles formas: consultar la Ayuda
svds(S)	Calcula algunos valores singulares de una matriz rectangular. Esta función tiene muchas posibles formas: consultar la Ayuda
normest(S,tol)	Estimación de la norma-2 con una determinada tolerancia (por defecto 1e-06)
condest(S)	Estimación de condición numérica sub-1
sprank(S)	Calcula el rango de una matriz dispersa
symbfact(S)	*Symbolic factorization analysis*. Devuelve información sobre los elementos que se harán distintos de cero en la factorización de Cholesky, sin llegar a realizar dicha factorización

Los sistemas de ecuaciones con matrices dispersas se pueden resolver con ***métodos directos***, que son variantes de la eliminación gaussiana. El camino habitual de acceder a los métodos directos es a través de los operadores / y \, igual que para matrices llenas.

También se pueden utilizar ***métodos iterativos***, que tienen la ventaja de no cambiar ningún elemento de la matriz. Se trata de obtener soluciones aproximadas después de un número finito de pasos.

Se llama factorizaciones "incompletas" a aquéllas que no calculan la factorización exacta sino una aproximada, despreciando los elementos que se hacen distintos de cero pero tienen un valor pequeño. Aunque la factorización es incompleta y sólo aproximada, se puede hacer en mucho menos tiempo y para ciertas finalidades es suficiente. Estas factorizaciones incompletas se utilizan por ejemplo como pre-condicionadores de algunos métodos iterativos.

Las siguientes funciones son muy especializadas y aquí sólo se van a citar sus nombres (en inglés, tal como los utiliza MATLAB). Para más información recurrir al ***Help*** y a la bibliografía especializada.

pcg()	Resuelve un sistema de ecuaciones lineales por el método del Gradiente Conjugado Pre-condicionado (Preconditioned Conjugate Gradients Method). La matriz debe ser simétrica y positivo-definida
bicg()	BiConjugate Gradients Method. Similar al anterior para matrices cuadradas que no son simétricas y positivo-definidas
bicgstab()	BiConjugate Gradients Stabilized Method.

cgs()	Conjugate Gradients Squared Method
gmres()	Generalized Minimum Residual Method
qmr()	Quasi-Minimal Residual Method
spparms()	Establece los parámetros para las funciones que trabajan con matrices sparse (set parameters for sparse matrix routines)
spaugment()	Form least squares augmented system

5.5.4. REGLAS GENERALES PARA OPERAR CON MATRICES DISPERSAS

El criterio general para trabajar con matrices dispersas en MATLAB es que casi todas las operaciones matriciales estándar funcionan de la misma forma sobre matrices dispersas que sobre matrices llenas. De todas formas, existen algunos criterios particulares que conviene conocer y que se enuncian a continuación:

1. Las funciones que aceptan una matriz como argumento y devuelven un escalar o un vector siempre devuelven un vector lleno, aunque el argumento sea disperso
2. Las funciones que aceptan como argumentos escalares o vectores y devuelven matrices devuelven matrices llenas
3. Las funciones de un solo argumento que reciben una matriz y devuelven una matriz o vector conservan el carácter del argumento (disperso o lleno). Ej: ***chol()***, ***diag()***, ***max()***, ***sum()***
4. Las funciones binarias devuelven resultados dispersos si ambos argumentos son dispersos. Si un operando es lleno devuelven lleno, excepto si la operación conserva los elementos cero y distintos de cero (por ejemplo: .* y ./)
5. La concatenación de matrices con ***cat*** o corchetes [ ] produce resultados dispersos para operaciones mixtas
6. Sub-indexado de matrices; **S**(i,j) a la derecha de una asignación produce resultados dispersos, mientras que a la izquierda de una asignación (=) mantiene el tipo de almacenamiento de **S**.

5.5.5. PERMUTACIONES DE FILAS Y/O COLUMNAS EN MATRICES SPARSE

Para permutar las filas de una matriz se debe pre-multiplicar por una matriz de permutación **P**, que es una matriz que deriva de la matriz identidad **I** por permutación de filas y/o columnas. Así, el producto **P*S** permuta filas de la matriz **S**, mientras que **S*P'** permuta columnas.

Un vector de permutación **p** (que contiene una permutación de los números naturales *1:n*) actúa sobre las filas **S**(**p**,:) o columnas **S**(:,**p**). El vector de permutación **p** es más compacto y eficiente que la matriz de permutación **P**. Por eso casi siempre los resultados de permutaciones realizadas o a realizar se dan como vector **p** (excepto en la factorización LU). Las sentencias siguientes ilustran la relación entre la matriz **P** y el vector **p**.

```
>> I = speye(5);
>> p=[2,1,5,4,3]
p =
     2     1     5     4     3
P = I(p,:)      % para calcular la matriz P a partir del vector p
P =
   (2,1)        1
   (1,2)        1
   (5,3)        1
   (4,4)        1
   (3,5)        1
```

```
p = (P*(1:n)')' % para calcular el vector p a partir de la matriz P
p =
     2     1     5     4     3
```

Puede comprobarse que la inversa de **P** es **P'**. La función de reordenación ***symrcm(A)*** tiende a minimizar la banda de la matriz agrupando los elementos junto a la diagonal, y ***symmd(A)*** minimiza el fill-in o llenado de una matriz simétrica, mientras que ***colmmd(A)*** lo hace con una matriz no simétrica.

5.6. Clases y objetos

MATLAB dispone de herramientas necesarias para realizar una Programación Orientada a Objetos (POO) con muchas de las características disponibles en otros lenguajes como C++ y Java. Las variables miembro de una clase son los miembros de una ***estructura***, considerada en el apartado 5.3, a partir de la página 54. Las funciones miembro de la clase se definen en un directorio con el mismo nombre de la clase precedido por el carácter @. Dichas funciones pueden ser públicas y privadas. A diferencia de C++ y Java, las funciones miembro deben recibir el objeto al que se aplican como uno de los argumentos explícitos, y no mediante el operador punto (.). Existen también los conceptos de herencia y polimorfismo.

En "***Aprenda Matlab 7.0 como si estuviera en Segundo***" se proporcionará (cuando esté disponible) una introducción a la Programación Orientada a Objetos con MATLAB 7.0. En la versión online de los manuales (formato *.PDF) se contiene una excelente explicación sobre el tema, con numerosos ejemplos.

6. PROGRAMACIÓN DE MATLAB

Como ya se ha dicho varias veces –incluso con algún ejemplo– MATLAB es una aplicación que se puede programar muy fácilmente. De todas formas, como lenguaje de programación pronto verá que no tiene tantas posibilidades como otros lenguajes (ni tan complicadas...). Se comenzará viendo las bifurcaciones y bucles, y la lectura y escritura interactiva de variables, que son los elementos básicos de cualquier programa de una cierta complejidad.

6.1. Bifurcaciones y bucles

MATLAB posee un lenguaje de programación que –como cualquier otro lenguaje– dispone de sentencias para realizar ***bifurcaciones*** y ***bucles***. Las ***bifurcaciones*** permiten realizar una u otra operación según se cumpla o no una determinada condición. La Figura 29 muestra tres posibles formas de bifurcación.

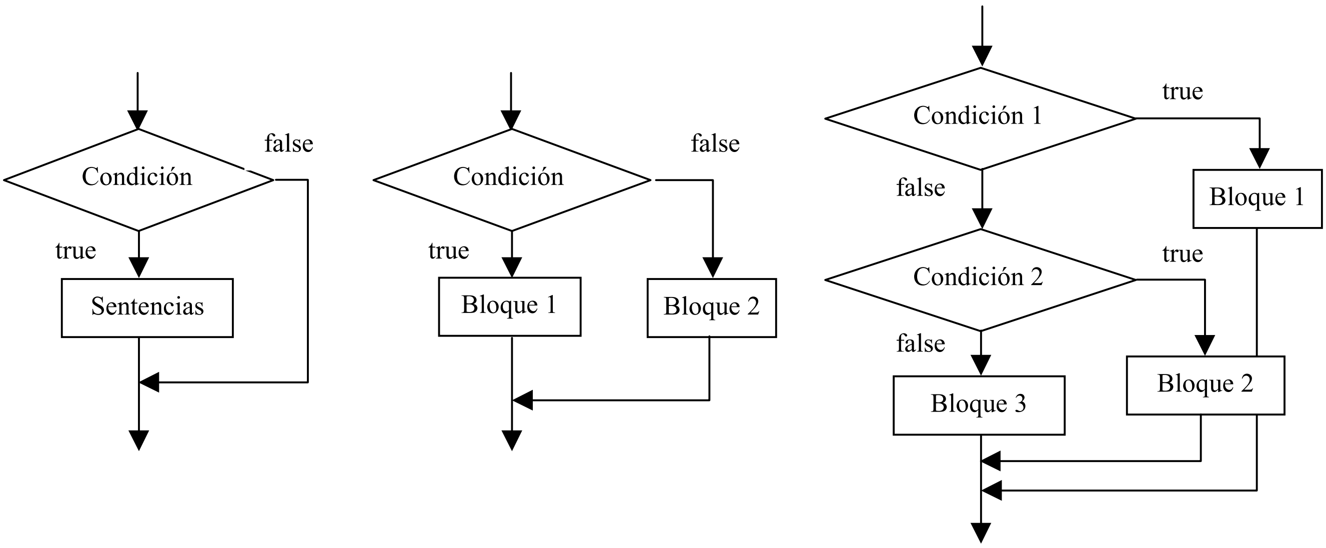


Figura 29. Ejemplos gráficos de bifurcaciones.

Los ***bucles*** permiten repetir las mismas o análogas operaciones sobre datos distintos. Mientras que en C/C++/Java el "cuerpo" de estas sentencias se determinaba mediante llaves {...}, en MATLAB se utiliza la palabra ***end*** con análoga finalidad. Existen también algunas otras diferencias de sintaxis.

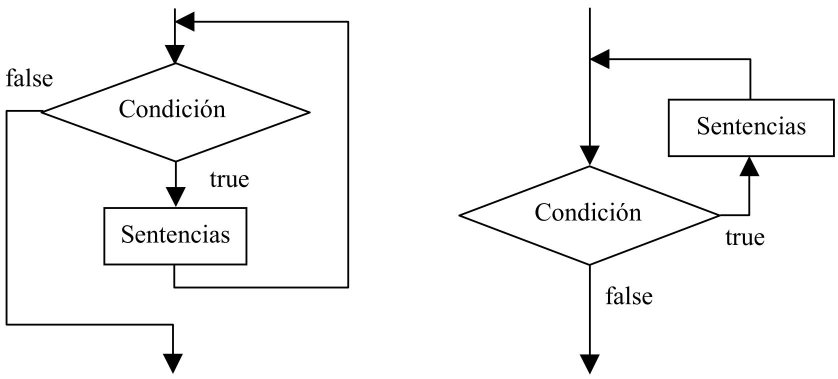


Figura 30. Bucles con control al principio y al final.

La Figura 30 muestra dos posibles formas de bucle, con el control situado al principio o al final del mismo. Si el control está situado al comienzo del bucle es posible que las sentencias no se ejecuten ninguna vez, por no haberse cumplido la condición cuando se llega al bucle por primera vez. Sin

embargo, si la condición está al final del bucle las sentencias se ejecutarán por lo menos una vez, aunque la condición no se cumpla. Muchos lenguajes de programación disponen de bucles con control al principio (***for*** y ***while*** en C/C++/Java) y al final (***do … while*** en C/C++/Java). En MATLAB no hay bucles con control al final del bucle, es decir, no existe construcción análoga a ***do ... while***.

Las bifurcaciones y bucles no sólo son útiles en la preparación de programas o de ficheros ***.m***. También se aplican con frecuencia en el uso interactivo de MATLAB, como se verá más adelante en algunos ejemplos.

6.1.1. SENTENCIA *IF*

En su forma más simple, la sentencia ***if*** se escribe en la forma siguiente (obsérvese que –a diferencia de C/C++/Java– la condición no va entre paréntesis, aunque se pueden poner si se desea)[15]:

```
if condicion
    sentencias
end
```

Existe también la *bifurcación múltiple*, en la que pueden concatenarse tantas condiciones como se desee, y que tiene la forma:

```
if condicion1
    bloque1
elseif condicion2
    bloque2
elseif condicion3
    bloque3
else  % opción por defecto para cuando no se cumplan las condiciones 1,2,3
    bloque4
end
```

donde la opción por defecto ***else*** puede ser omitida: si no está presente no se hace nada en caso de que no se cumpla ninguna de las condiciones que se han chequeado.

Una observación muy importante: la condición del ***if*** puede ser una *condición matricial*, del tipo **A**==**B**, donde **A** y **B** son matrices del mismo tamaño. Para que se considere que la *condición* se cumple, es necesario que sean *iguales dos a dos* ***todos*** *los elementos* de las matrices **A** y **B** ($a_{ij}=b_{ij}$, $1\leq i\leq m$, $1\leq j\leq n$). Basta que haya dos elementos a_{ij} y b_{ij} diferentes para que las matrices ya no sean iguales, y por tanto las sentencias del ***if*** no se ejecuten. Análogamente, una condición en la forma A~=B exige que todos los elementos sean diferentes dos a dos ($a_{ij}\neq b_{ij}$, $1\leq i\leq m$, $1\leq j\leq n$). Bastaría que hubiera dos elementos a_{ij} y b_{ij} iguales para que la condición no se cumpliese. En resumen:

if A==B	exige que ***todos*** los elementos sean ***iguales*** dos a dos
if A~=B	exige que ***todos*** los elementos sean ***diferentes*** dos a dos

Como se ha dicho, MATLAB dispone de funciones especiales para ayudar en el chequeo de condiciones matriciales. Por ejemplo, la función ***isequal(A, B)*** devuelve un uno si las dos matrices son idénticas y un cero en caso de que difieran en algo.

6.1.2. SENTENCIA *SWITCH*

La sentencia ***switch*** realiza una función análoga a un conjunto de ***if...elseif*** concatenados. Su forma general es la siguiente:

---

[15] En los ejemplos siguientes las ***sentencias*** aparecen desplazadas hacia la derecha respecto al ***if***, ***else*** o ***end***. Esto se hace así para que el programa resulte más legible, resultando más fácil ver dónde empieza y termina la bifurcación o el bucle. Es muy recomendable seguir esta práctica de programación.

```
switch switch_expresion
    case case_expr1,
        bloque1
    case {case_expr2, case_expr3, case_expr4,...}
        bloque2
    ...
    otherwise,    % opción por defecto
        bloque3
end
```

Al principio se evalúa la ***switch_expresion***, cuyo resultado debe ser un número escalar o una cadena de caracteres. Este resultado se compara con las ***case_expr***, y se ejecuta el bloque de sentencias que corresponda con ese resultado. Si ninguno es igual a ***switch_expresion*** se ejecutan las sentencias correspondientes a ***otherwise***. Según puede verse en el ejemplo anterior, es posible agrupar varias condiciones dentro de unas llaves (constituyendo lo que se llama un *cell array* o vector de celdas, explicado en el apartado 5.4); basta la igualdad con cualquier elemento del cell array para que se ejecute ese bloque de sentencias. La "igualdad" debe entenderse en el sentido del operador de igualdad (==) para escalares y la función ***strcmp()*** para cadenas de caracteres). A diferencia de C/C++/Java[16], en MATLAB sólo se ejecuta uno de los bloques relacionado con un ***case***.

6.1.3. SENTENCIA *FOR*

La sentencia ***for*** repite un conjunto de sentencias un número predeterminado de veces. La sentencia ***for*** de MATLAB es muy diferente y no tiene la generalidad de la sentencia ***for*** de C/C++/Java. La siguiente construcción ejecuta *sentencias* con valores de **i** de **1** a **n**, variando de uno en uno.

```
for i=1:n
    sentencias
end
```

o bien,

```
for i=vectorValores
    sentencias
end
```

donde **vectorValores** es un vector con los distintos valores que tomará la variable **i**.

En el siguiente ejemplo se presenta el caso más general para la variable del bucle (*valor_inicial: incremento: valor_final*); el bucle se ejecuta por primera vez con **i=n**, y luego **i** se va reduciendo de 0.2 en 0.2 hasta que llega a ser menor que 1, en cuyo caso el bucle se termina:

```
for i=n:-0.2:1
    sentencias
end
```

En el siguiente ejemplo se presenta una estructura correspondiente a dos ***bucles anidados***. La variable **j** es la que varía más rápidamente (por cada valor de **i**, **j** toma todos sus posibles valores):

```
for i=1:m
    for j=1:n
        sentencias
    end
end
```

Una última forma de interés del bucle ***for*** es la siguiente (**A** es una matriz):

```
for i=A
    sentencias
end
```

[16] En C se ejecuta el caso seleccionado y todos los siguientes, salvo que se utilice la sentencia ***break***.

en la que la variable **i** es un vector que va tomando en cada iteración el valor de una de las columnas de **A**.

Cuando se introducen interactivamente en la línea de comandos, los bucles ***for*** se ejecutan sólo después de introducir la sentencia ***end*** que los completa.

6.1.4. SENTENCIA *WHILE*

La estructura del bucle ***while*** es muy similar a la de C/C++/Java. Su sintaxis es la siguiente:

```
while condicion
    sentencias
end
```

donde ***condicion*** puede ser una expresión vectorial o matricial. Las *sentencias* se siguen ejecutando mientras haya elementos distintos de cero en ***condicion***, es decir, mientras haya algún o algunos elementos ***true***. El bucle se termina cuando *todos los elementos* de ***condicion*** son ***false*** (es decir, cero).

6.1.5. SENTENCIA *BREAK*

Al igual que en C/C++/Java, la sentencia ***break*** hace que se termine la ejecución del bucle ***for*** y/o ***while*** más interno de los que comprenden a dicha sentencia.

6.1.6. SENTENCIA *CONTINUE*

La sentencia ***continue*** hace que se pase inmediatamente a la siguiente iteración del bucle ***for*** o ***while***, saltando todas las sentencias que hay entre el ***continue*** y el fin del bucle en la iteración actual.

6.1.7. SENTENCIAS *TRY...CATCH...END*

La construcción ***try...catch...end*** permite gestionar los errores que se pueden producir en tiempo de ejecución. Su forma es la siguiente:

```
try
    sentencias1
catch
    sentencias2
end
```

En el caso de que durante la ejecución del bloque ***sentencias1*** se produzca un error, el control de la ejecución se transfiere al bloque ***sentencias2***. Si la ejecución transcurriera normalmente, ***sentencias2*** no se ejecutaría nunca. MATLAB dispone de una función ***lasterr*** que devuelve una cadena de caracteres con el mensaje correspondiente al último error que se ha producido. En la forma ***lasterr('')*** pone a cero este contador de errores, y hace que la función ***lasterr*** devuelva la matriz vacía [] hasta que se produzca un nuevo error.

6.2. Lectura y escritura interactiva de variables

Se verá a continuación una forma sencilla de leer variables desde teclado y escribir mensajes en la pantalla del PC. Más adelante se considerarán otros modos más generales –y complejos– de hacerlo.

6.2.1. FUNCIÓN *INPUT*

La función ***input*** permite imprimir un mensaje en la línea de comandos de MATLAB y recuperar como valor de retorno un valor numérico o el resultado de una expresión tecleada por el usuario. Después de imprimir el mensaje, el programa espera que el usuario teclee el valor numérico o la expresión. Cualquier expresión válida de MATLAB es aceptada por este comando. El usuario pue-

de teclear simplemente un vector o una matriz. En cualquier caso, la expresión introducida es evaluada con los valores actuales de las variables de MATLAB y el resultado se devuelve como valor de retorno. Véase un ejemplo de uso de esta función:

```
>> n = input('Teclee el número de ecuaciones')
```

Otra posible forma de esta función es la siguiente (obsérvese el parámetro `'s'`):

```
>> nombre = input('¿Cómo te llamas?','s')
```

En este caso el texto tecleado como respuesta se lee y se devuelve sin evaluar, con lo que se almacena en la cadena ***nombre***. Así pues, en este caso, si se teclea una fórmula, se almacena como texto sin evaluarse.

6.2.2. FUNCIÓN *DISP*

La función ***disp*** permite imprimir en pantalla un mensaje de texto o el valor de una matriz, pero sin imprimir su nombre. En realidad, ***disp*** siempre imprime vectores y/o matrices: las cadenas de caracteres son un caso particular de vectores. Considérense los siguientes ejemplos de cómo se utiliza:

```
>> disp('El programa ha terminado')
>> A=rand(4,4)
>> disp(A)
```

Ejecútense las sentencias anteriores en MATLAB y obsérvese la diferencia entre las dos formas de imprimir la matriz **A**.

6.3. Ficheros *.m

Los ficheros con extensión (***.m***) son ficheros de texto sin formato (ficheros ASCII) que constituyen el centro de la programación en MATLAB. Ya se han utilizado en varias ocasiones. Estos ficheros se crean y modifican con un editor de textos cualquiera. En el caso de MATLAB ejecutado en un PC bajo ***Windows***, lo mejor es utilizar su propio editor de textos, que es también ***Debugger***.

Existen dos tipos de ficheros ****.m***, los ***ficheros de comandos*** (llamados *scripts* en inglés) y las ***funciones***. Los primeros contienen simplemente un conjunto de comandos que se ejecutan sucesivamente cuando se teclea el nombre del fichero en la línea de comandos de MATLAB o se incluye dicho nombre en otro fichero ****.m***. Un fichero de comandos puede llamar a otros ficheros de comandos. Si un fichero de comandos se llama desde de la línea de comandos de MATLAB, las variables que crea pertenecen al ***espacio de trabajo base*** de MATLAB (recordar apartado 2.5.6), y permanecen en él cuando se termina la ejecución de dicho fichero.

Las ***funciones*** permiten definir funciones enteramente análogas a las de MATLAB, con su ***nombre***, sus ***argumentos*** y sus ***valores de retorno***. Los ficheros ****.m*** que definen funciones permiten extender las posibilidades de MATLAB; de hecho existen bibliotecas de ficheros ****.m*** que se venden (***toolkits***) o se distribuyen gratuitamente (a través de *Internet*). Las funciones definidas en ficheros ****.m*** se caracterizan porque la primera línea (que no sea un comentario) comienza por la palabra ***function***, seguida por los *valores de retorno* (entre corchetes [ ] y separados por comas, si hay más de uno), el signo igual (=) y el *nombre de la función*, seguido de los *argumentos* (entre paréntesis y separados por comas).

Recuérdese que un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso puede llamarse a sí mismo de forma recursiva. Los ficheros de comandos se pueden llamar también desde funciones, en cuyo caso las variables que se crean pertenecen al ***espacio de trabajo de la función***. El espacio de trabajo de una función es independiente del espacio de trabajo base y del espacio de trabajo de las demás funciones. Esto implica por ejemplo que no puede haber colisiones entre nombres de varia-

bles: aunque varias funciones tengan una variable llamada **A**, en realidad se trata de variables completamente distintas (a no ser que **A** haya sido declarada como variable ***global***).

A continuación se verá con un poco más de detalle ambos tipos de ficheros ***.m***.

6.3.1. FICHEROS DE COMANDOS (*SCRIPTS*)

Como ya se ha dicho, los ficheros de comandos o *scripts* son ficheros con un nombre tal como ***file1.m*** que contienen una sucesión de comandos análoga a la que se tecleaía en el uso interactivo del programa. Dichos comandos se ejecutan sucesivamente cuando se teclea el nombre del fichero que los contiene (sin la extensión), es decir cuando se teclea ***file1*** con el ejemplo considerado. Cuando se ejecuta desde la línea de comandos, las variables creadas por ***file1*** pertenecen al espacio de trabajo base de MATLAB. Por el contrario, si se ejecuta desde una función, las variables que crea pertenecen al espacio de trabajo de la función (ver apartado 2.5.6, en la página 14).

En los ficheros de comandos conviene poner los puntos y coma (;) al final de cada sentencia, para evitar una salida de resultados demasiado cuantiosa. Un fichero ***.m*** puede llamar a otros ficheros ***.m***, e incluso se puede llamar a sí mismo de modo recursivo. Sin embargo, no se puede hacer ***profile*** (ver apartado 6.11, en la página 95) de un fichero de comandos: sólo se puede hacer de las funciones.

Las variables definidas por los ficheros de comandos son variables del espacio de trabajo desde el que se ejecuta el fichero, esto es variables con el mismo carácter que las que se crean interactivamente en MATLAB si el fichero se ha ejecutado desde la línea de comandos. Al terminar la ejecución del *script*, dichas variables permanecen en memoria.

El comando ***echo*** hace que se impriman los comandos que están en un *script* a medida que van siendo ejecutados. Este comando tiene varias formas:

echo on	activa el ***echo*** en todos los ficheros script
echo off	desactiva el ***echo***
echo file on	donde 'file' es el nombre de un fichero de función, activa el ***echo*** en esa función
echo file off	desactiva el ***echo*** en la función
echo file	pasa de **on** a **off** y viceversa
echo on all	activa el ***echo*** en todas las funciones
echo off all	desactiva el ***echo*** de todas las funciones

Mención especial merece el fichero de comandos ***startup.m*** (ver apartado 2.7). Este fichero se ejecuta cada vez que se entra en MATLAB. En él puede introducir todos aquellos comandos que le interesa se ejecuten siempre al iniciar la sesión, por ejemplo ***format compact*** y los comandos necesarios para modificar el ***path***.

6.3.2. DEFINICIÓN DE FUNCIONES

La ***primera línea*** de un fichero llamado ***name.m*** que define una función tiene la forma:

```
function [lista de valores de retorno] = name(lista de argumentos)
```

donde ***name*** es el nombre de la función. Entre corchetes y separados por comas van los ***valores de retorno*** (siempre que haya más de uno), y entre paréntesis también separados por comas los ***argumentos***. Puede haber funciones sin valor de retorno y también sin argumentos. Recuérdese que los ***argumentos*** son los ***datos*** de la función y los ***valores de retorno*** sus ***resultados***. Si no hay valores de retorno se omiten los corchetes y el ***signo igual*** (=); si sólo hay un valor de retorno no hace falta poner corchetes. Tampoco hace falta poner paréntesis si no hay argumentos.

Una diferencia importante con C/C++/Java es que en MATLAB una función no puede modificar nunca los argumentos que recibe, de cara al entorno que ha realizado la llamada. Los resultados de una función de MATLAB se obtienen siempre a través de los valores de retorno, que pueden ser múltiples y matriciales. Tanto el número de argumentos como el de valores de retorno no tienen que ser fijos, dependiendo de cómo el usuario llama a la función[17].

Las variables definidas dentro de una función son ***variables locales***, en el sentido de que son inaccesibles desde otras partes del programa y en el de que no interfieren con variables del mismo nombre definidas en otras funciones o partes del programa. Se puede decir que pertenecen al propio espacio de trabajo de la función y no son vistas desde otros espacios de trabajo. Para que la función tenga acceso a variables que no han sido pasadas como argumentos es necesario declarar dichas variables como ***variables globales***, tanto en el programa principal como en las distintas funciones que deben acceder a su valor. Es frecuente utilizar el convenio de usar para las variables globales nombres largos (más de 5 letras) y con mayúsculas.

Por razones de eficiencia, los argumentos que recibe una función de MATLAB no se copian a variables locales si no son modificados por dicha función (en términos de C/C++ se diría que se pasan ***por referencia***). Esto tiene importantes consecuencias en términos de eficiencia y ahorro de tiempo de cálculo. Sin embargo, si dentro de la función se realizan modificaciones sobre los argumentos recibidos, antes se sacan copias de dichos argumentos a variables locales y se modifican las copias (diríase que en este caso los argumentos se pasan ***por valor***).

Dentro de la función, los valores de retorno deben ser calculados en algún momento (no hay sentencia ***return*** obligatoria, como en C/C++/Java). De todas formas, no hace falta calcular siempre todos los posibles valores de retorno de la función, sino sólo ***los que el usuario espera obtener*** en la sentencia de llamada a la función. En cualquier función existen dos variables definidas de modo automático, llamadas ***nargin*** y ***nargout***, que representan respectivamente el número de argumentos y el número de valores de retorno con los que la función ha sido llamada. Dentro de la función, estas variables pueden ser utilizadas como el programador desee.

La ejecución de una función termina cuando se llega a su última sentencia ejecutable. Si se quiere forzar el que una función termine de ejecutarse se puede utilizar la sentencia ***return***, que devuelve inmediatamente el control al entorno de llamada.

6.3.3. SENTENCIA *RETURN*

De ordinario las funciones devuelven el control después de que se ejecute la última de sus sentencias. La sentencia ***return***, incluida dentro del código de una función, hace que se devuelva inmediatamente el control al programa que realizó la llamada.

6.3.4. FUNCIONES CON NÚMERO VARIABLE DE ARGUMENTOS

Desde la versión 5.0, MATLAB dispone de una nueva forma de pasar a una función un número variable de argumentos por medio de la variable ***varargin***, que es un ***vector de celdas*** (ver apartado 5.4, en la página 56) que contiene tantos elementos como sean necesarios para poder recoger en dichos elementos todos los argumentos que se hayan pasado en la llamada. No es necesario que ***varargin*** sea el único argumento, pero sí debe ser el último de los que haya, pues recoge todos los argumentos a partir de una determinada posición. Recuérdese que a los elementos de un ***cell array*** se accede utilizando llaves {}, en lugar de paréntesis ().

---

17 Es un concepto distinto del de ***funciones sobrecargadas*** (funciones distintas con el mismo nombre y distintos argumentos), utilizadas en C/C++/Java. En MATLAB una misma función puede ser llamada con más o menos argumentos y valores de retorno. También en C/C++ es posible tener un número variable de argumentos, aunque no de valores de retorno.

De forma análoga, una función puede tener un número indeterminado de valores de retorno utilizando ***varargout***, que es también un ***cell array*** que agrupa los últimos valores de retorno de la función. Puede haber otros valores de retorno, pero ***varargout*** debe ser el último. El cell array ***varargout*** se debe crear dentro de la función y hay que dar valor a sus elementos antes de salir de la función. Recuérdese también que las variables ***nargin*** y ***nargout*** indican el número de argumentos y de valores de retorno con que ha sido llamada la función. A continuación se presenta un ejemplo sencillo: obsérvese el código de la siguiente función ***atan3***:

```
function varargout=atan3(varargin)
   if nargin==1
      rad = atan(varargin{1});
   elseif nargin==2
      rad = atan2(varargin{1},varargin{2});
   else
      disp('Error: más de dos argumentos')
      return
   end
   varargout{1}=rad;
   if nargout>1
      varargout{2}=rad*180/pi;
   end
```

MATLAB (y muchos otros lenguajes de programación) dispone de dos funciones, llamadas ***atan*** y ***atan2***, para calcular el arco cuya tangente tiene un determinado valor. El resultado de dichas funciones está expresado en radianes. La función ***atan*** recibe un único argumento, con lo cual el arco que devuelve está comprendido entre –π/2 y +π/2 (entre –90º y 90º), porque por ejemplo un arco de 45º es indistinguible de otro de –135, si sólo se conoce la tangente. La función ***atan2*** recibe dos argumentos, uno proporcional al seno del ángulo y otro al coseno. En este caso ya se pueden distinguir los ángulos en los cuatro cuadrantes, entre –π y π (entre –180º y 180º).

La función ***atan3*** definida anteriormente puede recibir uno o dos argumentos: si recibe uno llama a ***atan*** y si recibe dos llama a ***atan2*** (si recibe más da un mensaje de error). Además, ***atan3*** puede devolver uno o dos valores de retorno. Por ejemplo, si el usuario la llama en la forma:

```
>> a = atan3(1);
```

devuelve un valor de retorno que es el ángulo en radianes, pero si se llama en la forma:

```
>> [a, b] = atan3(1,-1);
```

devuelve dos valores de retorno, uno con el ángulo en radianes y otro en grados. Obsérvese cómo la función ***atan3*** utiliza los vectores de celdas ***varargin*** y ***varargout***, así como el número actual de argumentos ***nargin*** con los que ha sido llamada.

6.3.5. *HELP* PARA LAS FUNCIONES DE USUARIO

También las funciones creadas por el usuario pueden tener su ***help***, análogo al que tienen las propias funciones de MATLAB. Esto se consigue de la siguiente forma: las primeras líneas de comentarios de cada fichero de función son muy importantes, pues permiten construir un ***help*** sobre esa función. En otras palabras, cuando se teclea en la ventana de comandos de MATLAB:

```
>> help mi_func
```

el programa responde escribiendo las primeras líneas del fichero ***mi_func.m*** que comienzan por el carácter (%), es decir, que son comentarios.

De estas líneas, tiene una importancia particular la ***primera línea de comentarios*** (llamada en ocasiones línea H1). En ella hay que intentar poner la información más relevante sobre esa función. La razón es que existe una función, llamada ***lookfor*** que busca una determinada palabra en cada primera línea de comentario de todas las funciones ****.m***.

6.3.6. *HELP* DE DIRECTORIOS

MATLAB permite que los usuarios creen una ayuda general para todas las funciones que están en un determinado directorio. Para ello se debe crear en dicho directorio un fichero llamado ***contents.m***. A continuación se muestra un fichero típico ***contents.m*** correspondiente al directorio ***toolbox\local*** de MATLAB:

```
% Preferences.
%
% Saved preferences files.
%   startup     - User startup M-file.
%   finish      - User finish M-file.
%   matlabrc    - Master startup M-file.
%   pathdef     - Search path defaults.
%   docopt      - Web browser defaults.
%   printopt    - Printer defaults.
%
% Preference commands.
%   cedit       - Set command line editor keys.
%   terminal    - Set graphics terminal type.
%
% Configuration information.
%   hostid      - MATLAB server host identification number.
%   license     - License number.
%   version     - MATLAB version number.

% Utilities.
%   userpath    - User environment path.

% Copyright 1984-2002 The MathWorks, Inc.
% $Revision: 1.14 $  $Date: 2002/06/07 21:45:05 $
```

Compruébese que la información anterior es exactamente la que se imprime con el comando

```
>> help local
```

Si el fichero ***contents.m*** no existe, se listan las primeras líneas de comentarios (líneas H1) de todas las funciones que haya en ese directorio. Para que el ***Help*** de directorios funcione correctamente hace falta que ese directorio esté en el ***search path*** de MATLAB o que sea el directorio actual.

6.3.7. SUB-FUNCIONES

Tradicionalmente MATLAB obligaba a crear un fichero ****.m*** diferente por cada función. El nombre de la función debía coincidir con el nombre del fichero. A partir de la versión 5.0 se introdujeron las ***sub-funciones***, que son funciones adicionales definidas en un mismo fichero ****.m***, con nombres diferentes del nombre del fichero (y del nombre de la función principal) y que ***las sub-funciones sólo pueden ser llamadas por las funciones contenidas en ese fichero***, resultando "invisibles" para otras funciones externas.

A continuación se muestra un ejemplo contenido en un fichero llamado ***mi_fun.m***:

```
function y=mi_fun(a,b)
y=subfun1(a,b);


function x=subfun1(y,z)
x=subfun2(y,z);


function x=subfun2(y,z)
x=y+z+2;
```

6.3.8. FUNCIONES PRIVADAS

Las funciones privadas (***private***) son funciones que no se pueden llamar desde cualquier otra función, aunque se encuentren en el ***path*** o en el directorio actual. ***Sólo ciertas funciones están autorizadas a utilizarlas***. Las funciones privadas se definen en sub-directorios que se llaman ***private*** y sólo pueden ser llamadas por funciones definidas en el directorio padre del sub-directorio ***private***.

En la búsqueda de nombres que hace MATLAB cuando encuentra un nombre en una expresión, las funciones privadas se buscan inmediatamente después de las sub-funciones, y antes que las funciones de tipo general.

6.3.9. FUNCIONES *.P

Las funciones ****.p*** son funciones ****.m*** pre-compiladas con la función ***pcode***. Por defecto el resultado del comando ***pcode func.m*** es un fichero ***func.p*** en el directorio actual (el fichero ***func.m*** puede estar en cualquier directorio del ***search path***). El comando ***pcode -inplace func.m*** crea el fichero ***func.p*** en el mismo directorio donde encuentra el fichero ***func.m***. Pueden pasarse varios ficheros ****.m*** al comando ***pcode*** de una sola vez.

Los ficheros ****.p*** se ejecutan algo más rápidamente que los ****.m*** y permiten ocultar el código de los ficheros ASCII correspondientes a las funciones ****.m*** de MATLAB.

6.3.10. VARIABLES PERSISTENTES

Las ***variables persistentes*** son variables locales de las funciones (pertenecen al espacio de trabajo de la función y sólo son visibles en dicho espacio de trabajo), que ***conservan su valor*** entre distintas llamadas a la función. Por defecto, las variables locales de una función se crean y destruyen cada vez que se ejecuta la función. Las variables persistentes se pueden definir en funciones, pero no en ficheros de comandos. Es habitual utilizar para ellas letras mayúsculas. Las variables se declaran como persistentes utilizando la palabra ***persistent*** seguida de los nombres separados por blancos, como por ejemplo:

```
>> persistent VELOCIDAD TIEMPO
```

Las variables ***persistent*** se inicializan a la matriz vacía [] y permanecen en memoria hasta que se hace ***clear*** de la función o cuando se modifica el *fichero-M*. Para evitar que un fichero-M se modifique se puede utilizar el comando ***mlock file.m***, que impide la modificación del fichero. El comando ***munlock*** desbloquea el fichero mientras que la función ***mislocked*** permite saber si está bloqueado o no.

6.3.11. VARIABLES GLOBALES

Las variables globales son visibles en todas las funciones (y en el espacio de trabajo base o general) que las declaran como tales. Dichas variables de declaran precedidas por la palabra ***global*** y separadas por blancos, en la forma:

```
global VARIABLE1 VARIABLE2
```

Como ya se ha apuntado, estas variables sólo son visibles en los espacios de trabajo de las funciones que las declaran como tales (y en el propio espacio de trabajo base, si también ahí han sido declaradas como globales). Ya se ha dicho también que se suele recurrir al criterio de utilizar nombres largos y con mayúsculas, para distinguirlas fácilmente de las demás variables.

6.4. Referencias de función (*function handles*)

Las ***referencias de función*** (*function handles*) constituyen un nuevo mecanismo pera referirse a un nombre de función, introducido en MATLAB 6.0. En versiones anteriores la única forma de referir-

se a una función era por medio del ***nombre***. Téngase en cuenta que MATLAB, al igual que otros lenguajes de programación como C/C++ y Java, admite ***funciones sobrecargadas*** (*overloaded functions*), esto es, funciones diferentes que tienen el mismo nombre pero se diferencian entre sí por el número y tipo de los argumentos. Cuando un programa llama a una función sobrecargada, MATLAB analiza los tipos de los argumentos incluidos en la llamada y llama a la función que mejor se adapta a esos tipos de argumentos. Las ***referencias de función*** permiten al programador un mayor control sobre la función que es efectivamente llamada y tienen algunas otras ventajas que se verán en este apartado y en sus sub-apartados.

El principal uso de las ***referencias de función*** (como de los nombres de función en versiones anteriores) es el pasar a una función el nombre de otra función, junto con sus argumentos, para que la pueda ejecutar. A estas funciones que ejecutan otras funciones que se les pasan como argumentos se les llama ***funciones de función***, y se analizan con más detalle en el apartado 6.10, a partir de la página 83. Por ejemplo, MATLAB dispone de una función llamada ***quad*** que calcula la integral definida de una función entre unos límites dados. La función ***quad*** es genérica, esto es calcula, mediante métodos numéricos, integrales definidas de una amplia variedad de funciones, pero para que pueda calcular dicha integral hay que proporcionarle la función a integrar. Por ejemplo, para calcular la integral entre ***0*** y π de la función ***seno(x)*** se puede utilizar la sentencia:

```
>> area=quad('sin',0,pi)
area =
    2.0000
```

La función ***quad*** ejecuta la función ***sin*** por medio de la función ***feval***, que tiene la siguiente forma general:

```
feval(funcname, arg1, arg2, arg3, ...)
```

donde ***funcname*** es una cadena de caracteres con el nombre de la función a evaluar, y ***arg1***, ***arg2***, ***arg3***, ... son los argumentos que se le pasan a ***feval*** para que se los pueda pasar a ***funcname***.

A partir de la versión 6.0 de MATLAB ya no se pasa, a las funciones de función, el nombre de la función como en el ejemplo anterior, sino una ***referencia de función*** o *function handle*. De todas formas, para mantener la compatibilidad con los programas desarrollados en versiones anteriores, se sigue admitiendo que se pase a ***feval*** el nombre de la función, pero este mecanismo ya no se soportará en versiones posteriores. En MATLAB 6.* la forma correcta de ejecutar el ejemplo anterior sería (se explicará con más detalle a continuación):

```
fh=@sin;
area=quad(fh,0,pi);
```

donde la variable ***fh*** es una ***referencia de función***, que es un ***nuevo tipo de dato*** de MATLAB, con todas las posibilidades y limitaciones que esto supone.

6.4.1. CREACIÓN DE REFERENCIAS DE FUNCIÓN

Ya se ha comentado que las ***referencias de función*** son un nuevo tipo de datos de MATLAB 6. Una referencia de función se puede crear de dos formas diferentes:

1. Mediante el ***operador @*** ("at" o "arroba")

   La referencia a la función se crea precediendo el nombre de la función por el operador @. El resultado puede asignarse a una variable o pasarse como argumento a una función. Ejemplos:

   ```
   fh = @sin;
   area = quad(@sin, 0, pi);
   ```

2. Mediante la función ***str2func***

La función ***str2func*** recibe como argumento una cadena de caracteres conteniendo el nombre de una función y devuelve como valor de retorno la referencia de función. Una de las ventajas de esta función es que puede realizar la conversión de un vector de celdas con los nombres en un vector de referencias de función. Ejemplos:

```
>> fh = str2func('sin');
>> str2func({'sin','cos','tan'})
ans =
    @sin    @cos    @tan
```

Una característica común e importante de ambos métodos es que se aplican solamente al ***nombre de la función***, y no al nombre de la función precedido o cualificado por su ***path***. Además los nombres de función deben tener menos de 31 caracteres.

6.4.2. EVALUACIÓN DE FUNCIONES MEDIANTE REFERENCIAS

La principal aplicación de las ***referencias de función*** es pasar información de funciones a otras funciones para que aquéllas puedan ser ejecutadas por éstas. Para evaluar una referencia de función MATLAB utiliza la función ***feval***, que se llama de la forma siguiente:

```
[r1, r2, r3, ...] = feval(fh, arg1, arg2, arg3, ...)
```

donde ***fh*** es una referencia de función y ***r1***, ***r2***, ***r3***, ... y ***arg1***, ***arg2***, ***arg3***, ... son respectivamente los valores de retorno y los argumentos de la función cuya referencia es ***fh***.

Sobre la función ***feval*** hay que hacer dos observaciones:

1. La referencia de función ***fh*** debe ser un escalar. En otras palabras, no es posible evaluar un array de referencias de función con una sola llamada a ***feval***.
2. La función ***fh*** que se ejecuta en el momento de la llamada a ***feval*** depende de la situación en el momento en que se creó la referencia de función, y no de la situación en el momento de la llamada a ***feval***. Por ejemplo, si después de crear la referencia ***fh*** se cambia de directorio la función correspondiente, en el momento de la ejecución no será posible encontrarla; si después de crear ***fh*** se crea una nueva función con el mismo nombre, esta función no podrá nunca ser ejecutada por medio de la referencia creada previamente.

El siguiente ejemplo muestra cómo se puede ejecutar una ***sub-función*** desde otra función definida en un fichero ****.m*** diferente. Recuérdese que, en principio, las sub-funciones sólo son accesibles desde otras funciones definidas en el mismo fichero ****.m***. Supóngase que se crea un fichero llamado ***pruebafh.m*** que contiene las siguientes líneas (se define una función principal ***pruebafh*** que se llama como el fichero y una sub-función ***subf***):

```
% fichero pruebafh.m
function mifh=pruebafh
mifh=@subf;


function A=subf(B, C)
A=B+C;
```

Obsérvese que la función principal ***pruebafh*** devuelve una referencia a la sub-función ***subf***. En principio sólo ***pruebafh*** tiene acceso a ***subf*** y gracias a ese acceso puede crear la referencia ***mifh***. Sin embargo, una vez que la referencia a ***subf*** ha sido creada y devuelta como valor de retorno, cualquier función con acceso a ***pruebafh*** podrá también acceder a la sub-función ***subf***. El siguiente programa principal, definido en un fichero ***pruebafhMain.m***, puede acceder a la sub-función gracias a la referencia de función (si se intenta acceder directamente se obtiene un error).

```
% fichero pruebafhMain.m
fh=pruebafh
A=rand(3);
B=eye(3)*10;
C=feval(fh,A,B)
% D=subf(A,B) % ERROR
disp('Ya he terminado')
```

Este ejemplo sencillo es bastante significativo respecto a los beneficios que se pueden obtener de las referencias de función.

6.4.3. INFORMACIÓN CONTENIDA POR UNA REFERENCIA DE FUNCIÓN. FUNCIONES SOBRECARGADAS

Una referencia de función puede contener información de varias funciones, en concreto de todas aquellas funciones que fueran "visibles" en el momento en el que dicha referencia fue creada. Recuérdese que funciones visibles, además de las funciones intrínsecas de MATLAB (*built-in functions*) son las funciones que están definidas en el ***directorio actual*** y en los directorios definidos en el ***path*** de MATLAB.

La función ***functions*** permite obtener toda la información disponible de una referencia de función. Obsérvese la estructura salida del siguiente ejemplo (el campo ***methods*** es a su vez una estructura que puede mostrarse por separado):

```
>> info=functions(@deblank)
    function: 'deblank'
        type: 'overloaded'
        file: 'c:\matlab6p5\toolbox\matlab\strfun\deblank'
     methods: [1x1 struct]
>> info.methods
ans =
    cell: 'c:\matlab6p5\toolbox\matlab\strfun\@cell\deblank'
```

En este caso concreto se ha considerado la función ***deblank***, que permite eliminar caracteres en blanco en cadenas de caracteres o en vectoras de celdas con cadenas de caracteres. El valor de retorno de la función ***functions*** es una estructura con los cuatro campos siguientes:

function	cadena de caracteres con el nombre de la función a la que corresponde la referencia
type	Cadena de caracteres con uno de los siguientes valores: *'simple'*, *'subfunction'*, *'private'*, *'constructor'* y *'overloaded'*.
file	Cadena de caracteres que contiene el nombre del fichero ****.m*** en el que está definida la función o bien el texto '*MATLAB built-in functio*n'.
methods	Estructura que contiene los ***paths*** de los ficheros ****.m*** en los que están definidas las funciones sobrecargadas que se corresponden con esta referencia.

El argumento de la función ***functions*** debe ser una referencia de función ***escalar*** (no puede ser un array de referencias de función).

Los distintos valores del campo ***type*** tienen los siguientes significados:

simple	Funciones intrínsecas no sobrecargadas.
overloaded	Funciones sobrecargadas. Son las únicas que tienen campo ***methods***.
constructor	Constructores relacionados con clases y objetos.
subfunction	Funciones definidas en un fichero ****.m*** de otra función.
private	Funciones privadas (definidas en un subdirectorio ***private***).

Las ***funciones sobrecargadas*** (*overloaded*) tienen un interés especial, pues son las únicas que tienen el campo ***methods*** y las que pueden dar origen a más dificultades o problemas. De modo análogo a otros lenguajes de programación como C/C++ y Java, las funciones sobrecargadas de MATLAB son funciones que tienen el mismo nombre, pero distintos tipos de argumentos y distinto código (en otras palabras, funciones diferentes que sólo coinciden en el nombre).

Las funciones ***default*** son las que no tienen argumentos especializados. Otras funciones esperan recibir un argumento de un tipo más concreto. Salvo que haya una función especializada cuyos argumentos coincidan con los tipos de la llamada, MATLAB utilizará la función ***default***.

6.4.4. OTROS ASPECTOS DE LAS REFERENCIAS DE FUNCIÓN

De la misma manera que una cadena de caracteres pude ser convertida en una referencia de función por medio de la función ***str2func***, MATLAB dispone de la función ***func2str*** que realiza la conversión inversa. Puede ser interesante convertir una referencia de función en cadena de caracteres para construir mensajes de error en relación con construcciones ***try...catch***.

Otras funciones que pueden utilizarse en relación con las referencias de función son las siguientes (se presentas mediante ejemplos):

`isa(unaVariable, 'function_handle')`

Función que devuelve "1" ó "0" según ***unaVariable*** sea o no una referencia de función. Esta función se podría utilizar dentro de una función de función para comprobar que el argumento que indica la función a ejecutar con ***feval*** ha llegado correctamente.

`isequal(unfh, otrofh)`

Función que compara dos referencias a función e indica si dan acceso exactamente a los mismos métodos o no.

Como las referencias de función son variables estándar de MATLAB, pueden guardarse y recuperarse de una sesión a otra por medio de los comandos ***save*** y ***load***. Hay que tener cuidado al utilizar referencias de función creadas en sesiones anteriores, porque si ha cambiado el entorno de trabajo en algo que les afecte se obtendrá un error en tiempo de ejecución.

6.4.5. UTILIDAD DE LAS REFERENCIAS DE FUNCIÓN

La principal utilidad de las referencias de función es el pasar información de una función a otras funciones que la deben poder ejecutar por medio de ***feval***. Algunas otras ventajas de las referencias de funciones son las siguientes:

1. Se pueden encontrar todas las funciones con el mismo nombre que son visibles en un determinado estado del programa. De esta forma se tiene más control sobre la función que verdaderamente se va a ejecutar.

2. Acceder desde cualquier parte de un programa a las funciones privadas y a las sub-funciones. De esta forma se puede reducir el número de ficheros ****.m*** necesarios, pues muchas funciones se podrán definir como sub-funciones en un mismo fichero.

3. Mejorar la eficiencia de las funciones que se utilizan repetidamente, pues no es necesario buscar el fichero ****.m*** cada vez.

4. Las referencias de función son, como se ha dicho, variables ordinarias de MATLAB que pueden ser agrupadas en arrays de una o más dimensiones.

6.4.6. FUNCIONES INLINE

MATLAB permite definir funciones a partir de expresiones matemáticas por medio de la función ***inline***. Esta función trata de averiguar inteligentemente cuáles son los argumentos de la función ***inline***, a partir del contenido de la expresión matemática. Por defecto se supone que 'x' es el argummento, aunque es también posible determinarlos explícitamente al llamar a ***inline***. Considérense los siguientes ejemplos:

```
>> f=inline('expresión entre apóstrofos');
>> f=inline('expresión', a1, a2, a3); % los argumentos son 'a1', 'a2', 'a3'
>> f=inline('expresión', N);  % los argumentos son 'x', 'P1', ..., 'PN'
```

Las funciones ***inline*** se llaman con el handle (**f** en las sentencias anteriores) seguido de los argumentos entre paréntesis.

6.4.7. FUNCIONES ANÓNIMAS

Las funciones anónimas constituyen una forma muy flexible de crear funciones sobre la marcha, bien en la línea de comandos, bien en una línea cualquiera de una función o de un fichero ***.m***. La forma general de las funciones anónimas es la siguiente:

```
fhandle = @(argumentos) expresión;
```

Después de ser creada, la función anónima puede ser llamada a través del ***fhandle*** seguido de la lista de argumentos actuales entre paréntesis, o también puede ser pasada a otra función como argumento, también por medio del ***fhandle***. Por ejemplo, la siguiente función anónima calcula el valor del seno del ángulo doble:

```
senoAngDoble = @(ang) 2*sin(and).*cos(ang);
```

Las funciones anónimas acceden a las variables del espacio de trabajo en el que son definidas y crean en ese momento una copia de las variables que utilizan. El valor de dichas variables ya no se actualiza; sólo los argumentos pueden cambiar de valor. De esta forma, las funciones anónimas pueden ejecutarse a ravés del handle en otros espacios de trabajo.

Si las funciones anónimas no tienen argumentos hay que poner los paréntesis vacíos, tanto al definirlas como al llamarlas. Por otra parte, pueden tener varios valores de retorno cuando la expresion que contienen devuelve varios valores. Se pueden recoger todos o sólo algunos de estos valores.

La única limitación importante de las funciones anónimas es que están limitadas a ***una única expresión*** ejecutable de MATLAB. Por otra parte, su uso no tiene más limitaciones que las del uso del *handle* de una función cualquiera.

6.4.8. FUNCIONES ANIDADADAS

Las ***funciones anidadas*** (*nested functions*) son funciones definidas dentro de otras funciones, las llamadas ***funciones contenedoras***. Cuando se definen funciones anidadas es imprescindible terminar con una sentencia ***end*** la definición de cada una de las funciones contenidas en el fichero ***.m***.

Las funciones anidadas sirven para tener un mayor y mejor control sobre la visibilidad de las funciones (qué función puede ser llamada y desde dónde), así como sobre la visibilidad que dichas fiunciones tienen sobre las distintas variables del espacio de trabajo, incluyendo las que no les han sido pasadas como argumentos.

Una función puede contener varias funciones anidadas al mismo nivel, y una función anidada puede también ser contenedora, es decir contener a su vez una ó más funciones anidadas.

A continuación se muestra un ejemplo sencillo de múltiples funciones anidadas a varios niveles:

```
function A(arg1)
    ...
    function Aa(a1,a2)
        ...
        function Aa1()
            ...
        end
    end
    ....
    function Ab()
        ...
        function Ab1()
            ...
        end
        ...
        function Ab2()
            ...
        end
        ...
    end
    ....
end
```

Reglas para llamar a las funciones anidadas:

1. Una función anidada se puede llamar desde cualquier otra función que tenga acceso al handle.
2. Una función anidada se puede llamar directamente desde la función del nivel inmediatamente superior. Por ejemplo, la función **A** puede llamar a las funciones **Aa** y **Ab**, pero no a las funciones **Ab1** y **Ab2** que están dos niveles más abajo.
3. Una función anidada puede llamar a otras funciones anidadas en la misma función contenedora y al mismo nivel. Por ejemplo, la función **Aa** puede llamar a la función **Ab**, y la **Ab1** a la **Ab2**.
4. Una función anidada puede llamar a todas las que están por encima de ella en línea directa. Por ejemplo, la función **Ab2** puede llamar a las funciones **A** y **Ab**. Una función anidada no puede llamar a las que están al mismo nivel en otra rama; por ejemplo, **Aa1** no pueede llamar a **Ab2**.
5. Además, cualquier función anidada puede llamar a todas las sub-funciones definidas en el mismo fichero.

Reglas para la visibilidad de las variables con sub-funciones y funciones anidadas:

1. En general, las funciones y sub-funciones definidas en el mismo fichero tienen espacios de trabajo diferentes. Como consecuencia, las variables definidas en una función son ***variables locales*** que no pueden ser vistas por otras funciones o sub-funciones.
2. También una función anidada tiene su propio espacio de trabajo, pero además tiene acceso a todas las variables definidas por las funciones que están por encima de ella en la jerarquía de funciones anidadas.
3. También las funciones contenedoras ven y pueden modificar las variables locales definidas en sus funciones anidadas, siempre que definan o usen dichas variables. Como regla general, una variable usada o definida en una función anidada pertenece al espacio de trabajo de la función contenedora más exterior que de alguna manera accede a dicha variable.
4. Como consecuencia de lo dicho, si una función contenedora no usa o define una variable, pero esa variable es definida por dos funciones anidadas al mismo nivel en dicha función, dichas variables en las funciones anidadas son realmente variables distintas, pues al no ser usadas por la función contenedora no se transmiten a su espacio de trabajo.

5. Las variables correspondientes a los valores de retorno de una función anidada no pertenecen al espacio de trabajo de las funciones contenedoras que las llaman. Los valores d eretorno deben ser recogidoa explícitamente.

Recuérdese que el handle debe ser creado desde un punto del programa en el que la función es visible. Sin embargo, es posible utilizarlo luego en otro punto desde el que la función ya no sea visible. Estas reglas se aplican también a las funciones anidadas, aunque con algunas peculiaridades. En el momento de la creación del handle, las funciones anidadas tienen acceso a un espacio de trabajo ampliado con el de otras funciones del fichero ***.m**, según se ha expuesto. Para que esta función pueda ser llamada a través del handle en otro lugar del programa, en el momento de la creación del handle se crea una copia de las variables de su espacio de trabajo ampliado; estas copias son de tipo ***persistent*** y se conservan entre llamadas. Se recomienda ver los ejemplos en el ***Help*** de MATLAB.

6.5. Entrada y salida de datos

Ya se ha visto una forma de realizar la entrada interactiva de datos por medio de la función ***input*** y de imprimir resultados por medio de la función ***disp***. Ahora se van a ver otras formas de intercambiar datos con otras aplicaciones.

6.5.1. IMPORTAR DATOS DE OTRAS APLICACIONES

Hay varias formas de pasar datos de otras aplicaciones –por ejemplo de ***Excel***– a MATLAB. Se pueden enumerar las siguientes:

– se puede utilizar el ***Copy*** y ***Paste*** para copiar datos de la aplicación original y depositarlos entre los corchetes de una matriz o vector, en una línea de comandos de MATLAB. Tiene el inconveniente de que estos datos no se pueden editar.

– se puede crear un fichero ****.m*** con un editor de textos, con lo cual no existen problemas de edición.

– es posible leer un ***flat file*** escrito con caracteres ASCII. Un *flat file* es un fichero con filas de longitud constante separadas con ***Intro***, y varios datos por fila separados por *blancos*. Estos ficheros pueden ser leídos desde MATLAB con el comando ***load***. Si se ejecuta ***load datos.txt*** el contenido del ***flat file*** se deposita en una matriz con el nombre ***datos***. Por ejemplo, creando un fichero llamado ***flat.txt*** que contenga las líneas:

```
23.456  56.032 67.802
 3.749 -98.906 34.910
```

el comando **`A=load('flat.txt')`** leerá estos valores y los asignará a la matriz ***A***. Para más información utilizar ***help load***.

– el comando ***textread*** permite leer datos de cualquier tipo de un fichero siempre que estén convenientemente separados. Ver el ***Help*** para más información.

– se pueden leer datos de un fichero con las funciones ***fopen*** y ***fread*** (ver apartados 6.6.1 y 6.6.3, en las páginas 80 y 81).

– existen también otros métodos posibles: escribir funciones en C para traducir a formato ****.mat*** (y cargar después con ***load***), crear un fichero ejecutable ****.mex*** que lea los datos, etc. No se verán en estos apuntes.

6.5.2. EXPORTAR DATOS A OTRAS APLICACIONES

De forma análoga, también los resultados de MATLAB se pueden exportar a otras aplicaciones como ***Word*** o ***Excel***.

– utilizar el comando ***diary*** para datos de pequeño tamaño (ver apartado 2.9, en la página 21)
– utilizar el comando ***save*** con la opción ***–ascii*** (ver apartado 2.7, en la página 19)
– utilizar las funciones de bajo nivel ***fopen***, ***fwrite*** y otras (ver apartados 6.6.1 y 6.6.3, en las páginas 80 y 81)
– otros métodos que no se verán aquí: escribir subrutinas en C para traducir de formato ****.mat*** (guardando previamente con ***save***), crear un fichero ejecutable ****.mex*** que escriba los datos, etc.

Hay que señalar que los ficheros binarios ****.mat*** son trasportables entre versiones de MATLAB en distintos tipos de computadores, porque contienen información sobre el tipo de máquina en el *header* del fichero, y el programa realiza la transformación de modo automático. Los ficheros ****.m*** son de tipo ASCII, y por tanto pueden ser leídos por distintos computadores sin problemas de ningún tipo.

6.6. Lectura y escritura de ficheros

MATLAB dispone de funciones de lectura/escritura análogas a las del lenguaje C (en las que están inspiradas), aunque con algunas diferencias. En general son versiones simplificadas –con menos opciones y posibilidades– que las correspondientes funciones de C.

6.6.1. FUNCIONES *FOPEN* Y *FCLOSE*

Estas funciones sirven para abrir y cerrar ficheros, respectivamente. La función ***fopen*** tiene la forma siguiente:

```
[fi,texto] = fopen('filename','c')
```

donde **fi** es un valor de retorno que sirve como identificador del fichero, **texto** es un mensaje para caso de que se produzca un error, y **c** es un carácter (o dos) que indica el tipo de operación que se desea realizar. Las opciones más importantes son las siguientes:

'r'	lectura (de *read*)
'w'	escritura reemplazando (de *write*)
'a'	escritura a continuación (de *append*)
'r+'	lectura y escritura

Cuando por alguna razón el fichero no puede ser abierto, se devuelve un (-1). En este caso el valor de retorno **texto** puede proporcionar información sobre el tipo de error que se ha producido (también existe una función llamada ***ferror*** que permite obtener información sobre los errores. En el ***Help*** del programa se puede ver cómo utilizar esta función).

Después de realizar las operaciones de lectura y escritura deseadas, el fichero se puede cerrar con la función ***close*** en la forma siguiente:

```
st = fclose(fi)
```

donde **st** es un valor de retorno para posibles condiciones de error. Si se quieren cerrar a la vez todos los ficheros abiertos puede utilizarse el comando:

```
st = close('all')
```

6.6.2. FUNCIONES *FSCANF*, *SSCANF*, *FPRINTF* Y *SPRINTF*

Estas funciones permiten leer y escribir en ficheros ASCII, es decir, en ficheros formateados. La forma general de la función ***fscanf*** es la siguiente:

```
[var1,var2,...] = fscanf(fi,'cadena de control',size)
```

donde **fi** es el identificador del fichero (devuelto por la función ***fopen***), y **size** es un argumento opcional que puede indicar el tamaño del vector o matriz a leer. Obsérvese otra diferencia con C: las variables leídas se devuelven como valor de retorno y no como argumentos pasados por referencia (precedidos por el carácter &). La ***cadena de control*** va encerrada entre apóstrofos simples, y contiene los especificadores de formato para las variables:

%s	para cadenas de caracteres
%d	para variables enteras
%f	para variables de punto flotante
%lf	para variables de doble precisión

La función ***sscanf*** es similar a ***fscanf*** pero la entrada de caracteres no proviene de un fichero sino de una cadena de caracteres.

Finalmente, la función ***fprintf*** dirige su salida formateada hacia el fichero indicado por el identificador. Su forma general es:

```
fprintf(fi,'cadena de control',var1,var2,...)
```

Ésta es la función más parecida a su homóloga de C. La cadena de control contiene los formatos de escritura, que son similares a los de C, como muestran los ejemplos siguientes:

```
fprintf(fi,'El número de ecuaciones es: %d\n',n)
fprintf(fi,'El determinante es: %lf10.4\n',n)
```

De forma análoga, la función ***sprintf*** convierte su resultado en una cadena de caracteres que devuelve como valor de retorno, en vez de enviarlo a un fichero. Véase un ejemplo:

```
resultado = sprintf('El cuadrado de %f es %12.4f\n',n,n*n)
```

donde **resultado** es una cadena de caracteres. Esta función constituye el método más general de convertir números en cadenas de caracteres, por ejemplo para ponerlos como títulos de figuras.

6.6.3. FUNCIONES *FREAD* Y *FWRITE*

Estas funciones son análogas a ***fscanf*** y ***fprintf***, pero en vez de leer o escribir en un fichero de texto (ASCII), lo hacen en un *fichero binario*, no legible directamente por el usuario. Aunque dichos ficheros no se pueden leer y/o modificar con un editor de textos, tienen la ventaja de que las operaciones de lectura y escritura son mucho más rápidas, eficientes y precisas (no se pierden decimales al escribir). Esto es particularmente significativo para grandes ficheros de datos. Para más información sobre estas funciones se puede utilizar el ***help***.

6.6.4. FICHEROS DE ACCESO DIRECTO

De ordinario los ficheros de disco se leen y escriben secuencialmente, es decir, de principio a final, sin volver nunca hacia atrás ni realizar saltos. Sin embargo, a veces interesa acceder a un fichero de un modo arbitrario, sin ningún orden preestablecido. Esto se puede conseguir con las funciones ***ftell*** y ***fseek***.

En cada momento, hay una especie de *cursor* que indica en qué parte del fichero se está posicionado. La función ***fseek*** permite mover este cursor hacia delante o hacia atrás, respecto a la posición actual ('cof'), respecto al principio ('bof') o respecto al final del fichero ('eof'). La función ***ftell*** indica en qué posición está el cursor. Si alguna vez se necesita utilizar este tipo de acceso a disco, se puede buscar más información por medio del ***help***.

6.7. Recomendaciones generales de programación

Las funciones vectoriales de MATLAB son mucho más rápidas que sus contrapartidas escalares. En la medida de lo posible es muy interesante vectorizar los algoritmos de cálculo, es decir, realizarlos con vectores y matrices, y no con variables escalares dentro de bucles.

Aunque los vectores y matrices pueden ir creciendo a medida que se necesita, es mucho más rápido reservarles toda la memoria necesaria al comienzo del programa. Se puede utilizar para ello la función ***zeros***. Además de este modo la memoria reservada es contigua.

Es importante utilizar el ***profile*** para conocer en qué sentencias de cada función se gasta la mayor parte del tiempo de cálculo. De esta forma se descubren "cuellos de botella" y se pueden desaroollar aplicaciones mucho más eficientes.

Conviene desarrollar los programas incrementalmente, comprobando cada función o componente que se añade. De esta forma siempre se construye sobre algo que ya ha sido comprobado y que funciona: si aparece algún error, lo más probable es que se deba a lo último que se ha añadido, y de esta manera la búsqueda de errores está acotada y es mucho más sencilla. Recuérdese que de ordinario el tiempo de corrección de errores en un programa puede ser 4 ó 5 veces superior al tiempo de programación. El ***debugger*** es una herramienta muy útil a la hora de acortar ese tiempo de puesta a punto.

En este mismo sentido, puede decirse que pensar bien las cosas al programar (sobre una hoja de papel en blanco, mejor que sobre la pantalla del PC) siempre es rentable, porque se disminuye más que proporcionalmente el tiempo de depuración y eliminación de errores.

Otro objetivo de la programación debe ser mantener el código lo más sencillo y ordenado posible. Al pensar en cómo hacer un programa o en cómo realizar determinada tarea es conveniente pensar siempre primero en la solución más sencilla, y luego plantearse otras cuestiones como la eficiencia.

Finalmente, el código debe ser escrito de una manera clara y ordenada, introduciendo comentarios, utilizando líneas en blanco para separar las distintas partes del programa, sangrando las líneas para ver claramente el rango de las bifurcaciones y bucles, utilizando nombres de variables que recuerden al significado de la magnitud física correspondientes, etc.

En cualquier caso, la mejor forma (y la única) de aprender a programar es programando.

6.8. Acelerador JIT (Just In Time) en MATLAB

La versión 6.5 de MATLAB incorporó por primera vez un acelerador para los ficheros ***.m***, que permite acercarse a las velocidades de otros lenguajes de programación como Fortran y C/C++.

Es importante saber qué tipo de programas pueden ser acelerados y cuáles no lo son. En principio MATLAB acelera los bucles de los ficheros ***.m*** que no contienen cierto tipo de sentencias. Más en concreto, se pueden ofrecer las siguientes recomendaciones prácticas:

1. No utilizar estructuras, vectores de celdas, clases ni llamadas a función por medio de referencias.
2. No utilizar hipermatrices con más de tres dimensiones.
3. Utilizar sólo llamadas a funciones nativas de MATLAB (funciones compiladas, no definidas por medio de ficheros ***.m*** o ficheros MEX).
4. No utilizar variables que cambian de tipo a lo largo de la ejecución (por ejemplo, una variable que primero es una cadena de caracteres y luego pasa a ser una matriz real).

5. No utilizar las variables **i** y/o **j** con números complejos como si fueran variables normales (por ejemplo, no utilizar **3*i**, sino **3i**).
6. No utilizar vectores y/o matrices que crecen a lo largo de la ejecución del programa. Es mucho mejor reservar previamente toda la memoria necesaria mediante la función ***zeros***, ***ones*** o equivalente.

6.9. Llamada a comandos del sistema operativo y a otras funciones externas

Estando en la ventana de comandos de MATLAB, se pueden ejecutar comandos de MS-DOS precediéndolos por el carácter (!), como por ejemplo:

```
>> !edit fichero1.m
```

Si el comando va seguido por el carácter ampersand (&) el comando se ejecuta en "background", es decir, se recupera el control del programa sin esperar que el comando termine de ejecutarse. Por ejemplo, para arrancar ***Notepad*** en background,

```
>> !notepad &
```

Existe también la posibilidad de arrancar una aplicación y dejarla iconizada. Esto se hace postponiendo el carácter barra vertical (|), como por ejemplo en el comando:

```
>> !notepad |
```

Algunos comandos de MATLAB realizan la misma función que los comandos análogos del sistema operativo MS-DOS, con lo que se puede evitar utilizar el operador (!). Algunos de estos comandos son los siguientes:

dir	contenido del directorio actual
what	ficheros ****.m*** en el directorio actual
delete filename	borra el fichero llamado ***filename***
mkdir(nd)	crea un sub-directorio con el nombre ***nd***
copyfile(sc, dst)	copia el fichero ***sc*** en el fichero ***dst***
type file.txt	imprime por la pantalla el contenido del fichero de texto ***file.txt***
cd	cambiar de directorio activo
pwd	muestra el path del directorio actual
which func	localiza una función llamada ***func***
lookfor palabra	busca ***palabra*** en todas las primeras líneas de los ficheros ****.m***

6.10. Funciones de función

Como ya se ha comentado al hablar de las referencias de función, en MATLAB existen funciones a las que hay que pasar como argumento el nombre de otras funciones, para que puedan ser llamadas desde dicha función. Así sucede por ejemplo si se desea calcular la integral definida de una función, resolver una ecuación no lineal, o integrar numéricamente una ecuación diferencial ordinaria (problema de valor inicial). Estos serán los tres casos –de gran importancia práctica– que se van a ver a continuación. Se comenzará por medio de un ejemplo, utilizando una función llamada ***prueba*** que se va a definir en un fichero llamado ***prueba.m***.

Para definir esta función, se debe elegir ***FILE/New/M-File*** en el menú de MATLAB. Si las cosas están "en orden" se abrirá el ***Editor&Debugger*** para que se pueda editar ese fichero. Una vez abierto el ***Editor***, se deben teclear las 2 líneas siguientes:

```
function y=prueba(x)
y = 1./((x-.3).^2+.01)+1./...
    ((x-.9).^2+.04)-6;
```

guardándolo después con el nombre de ***prueba.m***. La definición de funciones se ha visto con detalle en el apartado 6.3.2, a partir de la página 68. El fichero anterior ha definido una nueva función que puede ser utilizada como cualquier otra de las funciones de MATLAB. Antes de seguir adelante, conviene ver el aspecto que tiene esta función que se acaba de crear. Para dibujar la función ***prueba***, tecléense los siguientes comandos:

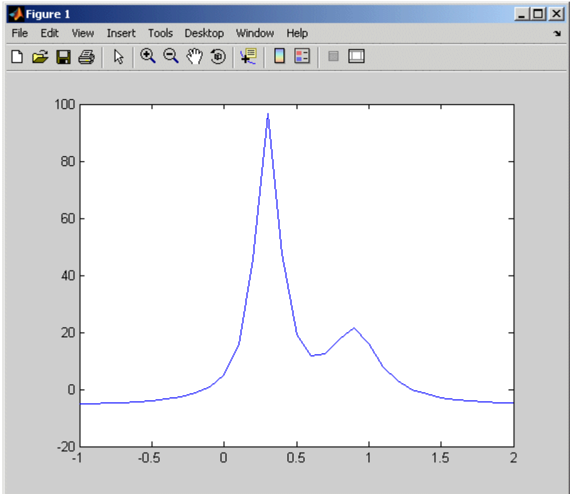


Figura 31. Función "prueba".

```
>> x=-1:0.1:2;
>> plot(x,prueba(x))
```

El resultado aparece en la Figura 31. Ya se está en condiciones de intentar hacer cálculos y pruebas con esta función.

6.10.1. Integración numérica de funciones

Lo primero que se va a hacer es calcular la integral definida de esta función entre dos valores de la abscisa **x**. En inglés, al cálculo numérico de integrales definidas se le llama *quadrature*. Sabiendo eso, no resulta extraño el comando con el cual se calcula el área comprendida bajo la función entre los puntos 0 y 1 (obsérvese que la referencia de la función a integrar se pasa por medio del operador @ precediendo al nombre de la función. También podría crearse una variable para ello):

```
>> area = quad(@prueba, 0, 1)
area =
   29.8583
```

Si se teclea ***help quad*** se puede obtener más de información sobre esta función, incluyendo el método utilizado (*Simpson*) y la forma de controlar el error de la integración.

La función ***quadl()*** utiliza un método de orden superior (*Lobatto*), mientras que la función ***dblquad()*** realiza integrales definidas dobles y la función ***triplequad()*** realiza integrales de volumen. Ver el ***Help*** o los manuales online para más información.

6.10.2. Ecuaciones no lineales y optimización

Después de todo, calcular integrales definidas no es tan difícil. Más difícil es desde luego calcular las raíces de ecuaciones no lineales, y el mínimo o los mínimos de una función. MATLAB dispone de las tres funciones siguientes:

fzero	calcula un cero o una raíz de una función de una variable
fminbnd	calcula el mínimo de una función de una variable
fminsearch	calcula el mínimo de una función de varias variables
optimset	permite establecer los parámetros del proceso de cálculo

Se empezará con el cálculo de raíces. Del gráfico de la función ***prueba*** entre -1 y 2 resulta evidente que dicha función tiene dos raíces en ese intervalo. La función ***fzero*** calcula una y se conforma:

¿Cuál es la que calcula? Pues depende de un parámetro o argumento que indica un punto de partida para buscar la raíz. Véanse los siguientes comandos y resultados:

```
>> fzero(@prueba, -.5)
ans =
   -0.1316
>> fzero(@prueba, 2)
ans =
    1.2995
```

En el primer caso se ha dicho al programa que empiece a buscar en el punto -0.5 y la solución encontrada ha sido -0.1316. En el segundo caso ha empezado a buscar en el punto de abscisa 2 y ha encontrado otra raíz en el punto 1.2995. Se ven claras las limitaciones de esta función.

La función ***fzero()*** tiene también otras formas interesantes:

fzero(@prueba, [x1,x2])	calcula una raíz en el intervalo x1-x2. Es necesario que la función tenga distinto signo en los extremos del intervalo.
fzero(@prueba, x, options)	calcula la raíz más próxima a **x** con ciertas opciones definidas en la estructura **options**. Esta estructura se crea con la función ***optimset***.

La función ***optimset*** tiene la siguiente forma general:

```
options = optimset('param1',val1,'param2',val2,...
```

en la que se indican los nombres de los parámetros u opciones que se desean modificar y los valores que se desea dar para cada uno de dichos parámetros. Una segunda forma general es:

```
options = optimset(oldopts, 'param1',val1,'param2',val2,...)
```

en la que se obtienen unas nuevas opciones modificando unas opciones anteriores con una serie de parejas *nombre-valor* de parámetros.

Existen muchas opciones que pueden ser definidas por medio de la función ***optimset***. Algunas de las más características son las siguientes (las dos primeras están dirigidas a evitar procesos iterativos que no acaben nunca y la tercera a controlar la precisión en los cálculos):

MaxFunEvals	máximo número de evaluaciones de función permitidas
MaxIter	máximo número de iteraciones
TolX	error máximo permitido en la abscisa de la raíz

Ahora se va a calcular el mínimo de la función ***prueba***. Defínase una función llamada ***prueba2*** que sea ***prueba*** cambiada de signo, y trátese de reproducir en el PC los siguientes comandos y resultados (para calcular máximos con ***fmin*** bastaría con cambiar el signo de la función):

```
>> plot(x,prueba2(x))
>> fminbnd(@prueba2, -1,2)
ans =
    0.3004
>> fminbnd(@prueba2, 0.5,1)
ans =
    0.8927
```

También a la función ***fminbnd*** se le puede pasar la estructura ***options***. Por ejemplo, para fijar un error de 10^{-08} se puede proceder del siguiente modo:

```
>> options=optimset('TolX', 1e-08);
>> fminbnd(@prueba2, 0.5,1, options)
```

En cualquier caso, es importante observar que para calcular las raíces o los valores mínimos de una función, hay que pasar el nombre de esta función como argumento a la función de MATLAB que va a hacer los cálculos. En esto consiste el concepto de ***función de función***.

MATLAB tiene un *toolbox* o paquete especial (que debe ser adquirido aparte)) con muchas más funciones orientadas a la *optimización*, es decir al cálculo de valores mínimos de funciones, con o sin restricciones.

6.10.3. INTEGRACIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES ORDINARIAS

Este es otro campo en el que las capacidades de MATLAB pueden resultar de gran utilidad a los ingenieros o futuros ingenieros interesados en la ***simulación***. MATLAB es capaz de calcular la evolución en el tiempo de sistemas de ecuaciones diferenciales ordinarias de primer orden, lineales y no lineales. Por el momento se supondrá que las ecuaciones diferenciales se pueden escribir en la forma:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t) \tag{7}$$

donde t es la variable escalar, y tanto **y** como su derivada son vectores. Un ejemplo típico puede ser el ***tiro parabólico***, considerando una resistencia del aire proporcional al cuadrado de la velocidad. Se supone que dicha fuerza responde a la siguiente expresión vectorial:

$$\begin{Bmatrix} F_x \\ F_y \end{Bmatrix} = -c\sqrt{\left(\dot{x}^2 + \dot{y}^2\right)}\begin{Bmatrix} \dot{x} \\ \dot{y} \end{Bmatrix} \tag{8}$$

donde c es una constante conocida. Las ecuaciones diferenciales del movimiento serán:

$$\begin{Bmatrix} \ddot{x} \\ \ddot{y} \end{Bmatrix} = \frac{1}{m}\left(\begin{Bmatrix} 0 \\ -mg \end{Bmatrix} - c\sqrt{\left(\dot{x}^2 + \dot{y}^2\right)}\begin{Bmatrix} \dot{x} \\ \dot{y} \end{Bmatrix}\right) \tag{9}$$

pero éste es un sistema de 2 ecuaciones diferenciales de orden 2. Para poderlo integrar debe tener la forma del sistema (7), y para ello se va a trasformar en un sistema de 4 ecuaciones diferenciales de primer orden, de la forma siguiente:

$$\begin{Bmatrix} \dot{u} \\ \dot{v} \\ \dot{x} \\ \dot{y} \end{Bmatrix} = \begin{Bmatrix} 0 \\ -g \\ u \\ v \end{Bmatrix} - \frac{c}{m}\sqrt{\left(u^2 + v^2\right)}\begin{Bmatrix} u \\ v \\ 0 \\ 0 \end{Bmatrix} \tag{10}$$

MATLAB dispone de varias funciones para integrar sistemas de ecuaciones diferenciales ordinarias de primer orden, entre ellas ***ode23***, que utiliza el método de *Runge-Kutta* de segundo/tercer orden, y ***ode45***, que utiliza el método de *Runge-Kutta-Fehlberg* de cuarto/quinto orden. Ambas exigen al usuario escribir una función que calcule las derivadas a partir del vector de variables, en la forma indicada por la ecuación (7).

Cree con el ***Editor/Debugger*** un fichero llamado ***tiropar.m*** que contenga las siguientes líneas:

```
function deriv=tiropar(t,y)
fac=-(0.001/1.0)*sqrt((y(1)^2+y(2)^2));
deriv=zeros(4,1);
deriv(1)=fac*y(1);
deriv(2)=fac*y(2)-9.8;
deriv(3)=y(1);
deriv(4)=y(2);
```

En el programa anterior se han supuesto unas constantes con los valores de *c*=0.001, *m*=1 y *g*=9.8. Falta fijar los valores iniciales de posición y velocidad. Se supondrá que el proyectil parte del origen con una velocidad de 100 m/seg y con un ángulo de 30º, lo que conduce a los valores iniciales siguientes: *u*(0)=100*cos(pi/6), *v*(0)=100*sin(pi/6), *x*(0)=0, *y*(0)=0. Los comandos para realizar la integración son los siguientes (se suponen agrupados en un fichero ***tiroparMain.m***):

```
% fichero tiroparMain.m
% intervalo de integración
tspan=[0,9];
% condiciones iniciales
y0=[100*cos(pi/6) 100*sin(pi/6) 0 0]';
% llamar a la función de integración numérica
[t,Y]=ode45(@tiropar,tspan,y0);
% dibujo de la altura en función del tiempo
plot(t,Y(:,4)), grid
disp('Ya he terminado')
```

En estos comandos **tspan** es un vector que define el intervalo temporal de integración. Es muy importante que en la función ***ode45***, el vector de condiciones iniciales **y0** sea un vector columna. El vector **t** devuelto por ***ode45*** contiene los valores del tiempo para los cuales se ha calculado la posición y velocidad. Dichos valores son controlados por la función ***ode45*** y no por el usuario, por lo que de ordinario no estarán igualmente espaciados. La matriz de resultados **Y** contiene cuatro columnas (las dos velocidades y las dos coordenadas de cada posición) y tantas filas como elementos tiene el vector **t**. En la Figura 32 se muestra el resultado del ejemplo anterior (posición vertical en función del tiempo).

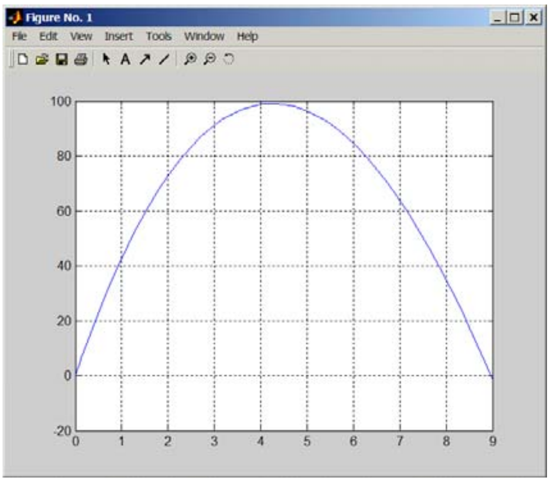


Figura 32. Tiro parabólico (posición vertical en función del tiempo).

MATLAB dispone de varias funciones para la integración de sistemas de ecuaciones diferenciales ordinarias. Se pueden citar las siguientes, clasificadas según una característica de las ecuaciones que se desea integrar:

Sistemas no-rígidos	***ode23***, ***ode45*** y ***ode113***
Sistemas rígidos	***ode15s***, ***ode23s***, odq23t y ***ode23tb***

La ***rigidez*** (*stiffness*, en la literatura inglesa) es una característica de muchos sistemas de ecuaciones diferenciales ordinarias que aparecen en la práctica y que los hace más difíciles de resolver. Una explicación detallada de esta característica excede la finalidad de este manual, pero sí se puede dar una muy breve explicación.

Muchos integradores numéricos están basados en fórmulas que permiten predecir el valor de la función en **t+Δt** a partir del valor de la función y de su derivada en el instante **t** y anteriores:

$$\mathbf{y}_{t+\Delta t} = \overline{f}\left(\mathbf{y}_t, \mathbf{y}_{t-\Delta t}, \ldots, \dot{\mathbf{y}}_t, \dot{\mathbf{y}}_{t-\Delta t}, \ldots, t\right) \quad (11)$$

A estos integradores se les llama ***integradores explícitos***. Todo lo que necesitan es que el usuario programe una función que calcule la derivada en la forma indicada en la ecuación (7).

En la solución de un sistema de ecuaciones diferenciales ordinarias aparecen combinadas diversas componentes oscilatorias (tipo seno, coseno o similar). Algunas de estas componentes oscilan más rápidamente que otras (tienen una frecuencia más elevada). Los problemas ***rígidos*** o ***stiff*** son aquellos en cuya solución participan componentes de frecuencias muy diferentes (muy altas y muy ba-

jas). Todos los integradores de MATLAB tienen control automático del error. Quiere esto decir que el usuario fija el error que está dispuesto a admitir en la solución y MATLAB ajusta el paso de la integración para conseguir ese error. Los integradores explícitos detectan la posible presencia de componentes de alta frecuencia en la solución y tratan de adaptar a ellas su paso, que se hace demasiado pequeño y termina por detener la integración.

Los ***integradores implícitos*** son mucho más apropiados para los problemas ***stiff***. En lugar de utilizar fórmulas del tipo de la ecuación (11) utilizan fórmulas del tipo:

$$\mathbf{y}_{t+\Delta t} = \overline{f}\left(\mathbf{y}_{t+\Delta t}, \mathbf{y}_t, \mathbf{y}_{t-\Delta t}, \ldots, \dot{\mathbf{y}}_{t+\Delta t}, \dot{\mathbf{y}}_t, \dot{\mathbf{y}}_{t-\Delta t}, \ldots, t\right) \qquad (12)$$

El problema con la expresión (12) es que para calcular la función en **t+Δt** hace uso de la derivada en ese mismo instante, que no puede ser conocida si no se conoce la función. Eso quiere decir que el sistema (12) es un ***sistema de ecuaciones no lineal***es que hay que resolver iterativamente. Los sistemas de ecuaciones no lineales se resuelven mucho más rápidamente si se conoce la derivada de la función (un ejemplo es el método de *Newton-Raphson*). Los integradores ***stiff*** de MATLAB son capaces de calcular esta derivada numéricamente (por diferencias finitas), pero son mucho más eficientes si el usuario es capaz de escribir una segunda función que les dé esta derivada. Esta derivada, que en realidad es una matriz de derivadas, es la ***Jacobiana***. Los integradores ***stiff***, además de la ecuación (7), permiten para el sistema de ecuaciones diferenciales una forma algo más especializada:

$$\mathbf{M}(\mathbf{y},t)\dot{\mathbf{y}} - \mathbf{f}(\mathbf{y},t) = \mathbf{0} \qquad (13)$$

en cuyo caso el usuario también tiene que proporcionar una función que calcule la matriz **M(y,*t*)**. La ecuación (13) representa un gran número de casos prácticos, por ejemplo los que surgen de las ecuaciones diferenciales del movimiento en Mecánica.

La forma más básica para todos los integradores de MATLAB es la siguiente:

```
[t, Y] = solvername(fh, tspan, y0)
```

donde **fh** es una referencia de la función que permite calcular la derivada según la expresión (7), **tspan** puede ser un vector de dos elementos [**tini, tfinal**] que representan el comienzo y el fin de la integración o un vector de tiempos [**tini:tstep:tfinal**] en los cuales se desea que MATLAB devuelva resultados, e **y0** es un vector columna con los valores iniciales. Como resultado se obtiene el vector **t** de tiempos en los que se dan resultados y una matriz **Y** con tantas filas como tiempos de salida y que representan cada una de ellas la salida en el correspondiente instante de tiempo.

Una forma más elaborada de llamar a los integradores de MATLAB es la siguiente:

```
[t, Y] = solvername(fh, tspan, y0, options)
```

donde ***options*** es una estructura similar a la vista en el apartado anterior para el cálculo de raíces y mínimos de funciones. En este caso la estructura ***options*** (que es diferente de la anterior, aunque se esté utilizando el mismo nombre) se determina por medio de la función ***odeset***, que admite las formas siguientes:

```
options = odeset('param1', val1,'param2', val2, ...);
options = odeset(oldopt, 'param1', val1,'param2', val2, ...);
```

Entre los parámetros u opciones más importantes se pueden citar los siguientes (se puede obtener más información sobre ellos consultando ***odeset*** en el ***Help***. Los *parámetros en cursiva* serán utilizados o explicados en los ejemplos que siguen):

Para el error	*RelTol* , *AbsTol*, NormControl
Para el paso	InitialStep, MaxStep

Para la matriz M	*Mass*, *MstateDependence*, *MvPattern*, MassSingular e InitialSlope
Para el Jacobiano	*Jacobian*, *JPattern*, *Vectorized*
Para la salida	*OutputFcn*, *OutputSel*, Refine, *Stats*

A continuación se va a repetir el ***ejemplo de tiro parabólico*** presentado al comienzo de esta Sección utilizando el integrador implícito ***ode15s*** con algunas opciones modificadas. Para ello la ecuación (10) se va a re-escribir en la forma de la ecuación (13), resultando:

$$\begin{bmatrix} m & 0 & 0 & 0 \\ 0 & m & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \dot{u} \\ \dot{v} \\ \dot{x} \\ \dot{y} \end{Bmatrix} = \begin{Bmatrix} 0 \\ -mg \\ u \\ v \end{Bmatrix} - c\sqrt{\left(u^2+v^2\right)} \begin{Bmatrix} u \\ v \\ 0 \\ 0 \end{Bmatrix} \tag{14}$$

En este caso el programa principal se ha denominado ***tiroparMain2*** y tiene la siguiente forma:

```
% fichero tiroparMain2.m
clear all, close all
t0=0; tf=10; npoints=51;
y0=[100*cos(pi/6),100*sin(pi/6),0,0]';
% vector de puntos en los que se desea resultados
tspan=[t0:(tf-t0)/(npoints-1):tf];
% modificación de las opciones por defecto
m=1; M=eye(4); M(1,1)=m; M(2,2)=m;
options = odeset('RelTol',1e-04, 'AbsTol',1e-06, ...
   'Stats','on', 'Mass',M, ...
   'OutputFcn',@odeplot, 'OutputSel',[1,2,3,4]);
% llamada a la función de integración numérica
[t,Y]=ode15s(@tiropar2,tspan,y0,options, m);
% dibujo de la altura del móvil en función del tiempo
figure, plot(t,Y(:,4)), grid
disp('Ya he terminado!')
```

Obsérvese cómo se han definido nuevas tolerancias para los errores absoluto y relativo, que se ha activado la opción de imprimir estadísticas y que se le indica al programa que se le da una matriz de masas constante en una variable llamada **M** (La otra opción para el argumento '**Mass**' es una referencia a la función que se encargará de calcular la matriz de masas). La función ***tiropar2*** no ha sufrido cambios importantes respecto a ***tiropar*** y es así:

```
function deriv=tiropar2(t,y, m)
% Ecuación diferencial en la forma
% M*yp=f(t,y);

deriv=zeros(4,1);
fac=-(0.001)*sqrt((y(1)^2+y(2)^2));
deriv(1)=fac*y(1);
deriv(2)=fac*y(2)-9.8*m;
deriv(3)=y(1);
deriv(4)=y(2);
```

El cambio más importante consiste en que a la función ***tiropar2*** se le ha pasado como argumento la msa **m** del proyectil. En efecto, la forma que tiene MATLAB para pasar argumentos desde el programa principal a las funciones llamadas por el integrador es poner estos argumentos a continuación de ***options***, en la llamada al integrador. El integrador recoge estos argumentos y los transmite.

El resultado de MATLAB incluye las estadísticas solicitadas y es el siguiente:

```
32 successful steps
0 failed attempts
58 function evaluations
1 partial derivatives
9 LU decompositions
51 solutions of linear systems
```

Una tercera y más sofisticada forma de llamar a los integradores de MATLAB podría ser la que se muestra a continuación. Aunque se trata de un ejemplo muy sencillo, se han incluido en él muchas de las características más avanzadas de los integradores de MATLAB, de aplicación en casos mucho más complicados. Los lectores interesados en las técnicas de simulación deben estudiar con atención el programa que sigue y los comentarios que se acompañan.

```
% fichero tiroparMain3.m
function tiroparMain3
% Intervalo de integracion
t0=0; tf=12; npoints=51;
tspan=[t0:(tf-t0)/(npoints-1):tf];
% condiciones iniciales
y0=[100*cos(pi/6),100*sin(pi/6),0,-10]';
% elementos ~=0 en la Jacobiana de f() en la ec. dif. M(t,y)*yp=f(t,y)
Jp=sparse([1 1 0 0; 1 1 0 0; 1 0 0 0; 0 1 0 0]);
options = odeset('RelTol',1e-06, 'AbsTol',1e-06, 'Mass',@tiropar3Masa, ...
   'MStateDep','none', 'OutputFcn',@tiropar3Salida, 'OutputSel',[3,4], ...
   'JPattern',Jp, 'Vectorized','on', 'Events',@tiropar3Eventos, 'Stats','on');
sol=ode15s(@tiropar3,tspan,y0,options,1,0.001);
% forma alternativa de llamar al integrador
% [T,Y, tEv, yEv, ev]=ode15s(@tiropar3,tspan,y0,options,1,0.001);
% resultados del cálculo de eventos
%  sol.xe  tiempos en los que se ha producido el evento
disp('Tiempos de corte con y(4)=0: '), disp(num2str(sol.xe));
%  sol.ye  vector de estado en los instantes del evento
disp('Valores del vector de estado en los eventos: '), disp(num2str(sol.ye));;
%  sol.ie  n´umero del evento que se ha producido
disp('Eventos que se han producido: '), disp(num2str(sol.ie));
T=tspan(find(tspan<sol.xe(2)));
% la funcion deval calcula la solucion en los tiempos deseados
Y=deval(sol,T);
% forma alternativa de llamar al integrador
disp('Ya he terminado!')

% calculo de la matriz de masas
function M=tiropar3Masa(t,m,c)
M=diag([m,m,1,1]);

% función para controlar la salida de resultados
function status=tiropar3Salida(t,y,flag,m,c)
% se llama a tiropar3Salida() en cada punto de salida y
%   esta función se encarga de llamar a odeplot
status=0;
switch flag
   case 'init'
      disp(['Entrando en salidaTiropar3 por primera vez']);
      odeplot(t,y,'init');
   case 'done'
      disp(['Entrando en salidaTiropar3 por última vez']);
      odeplot([],[],'done');
   case ''
      % puede haber resultados para más de un tiempo de salida
      for i=1:length(t)
         disp(['Entrando en salidaTiropar4 para t=',num2str(t(i))]);
         odeplot(t(i),y(:,i));
      end
end

```

```
53.  % funcion para controlar los eventos
54.  function [valor,esFinal,direccion]=tiropar3Eventos(t,y,m,c)
55.  if y(2)>0
56.     valor = y(4);   % unico valor que se controla
57.     esFinal = 0;    % no termina la integración al llegar al suelo
58.     direccion = 1;  % llegar al suelo en dirección ascendente
59.  else
60.     valor = y(4);
61.     esFinal = 1;    % termina la integración al llegar al suelo
62.     direccion = -1; % llegar al suelo en dirección descndente
63.  end
```

Por otra parte, el fichero ***tiropar3.m*** que evalúa la ecuación diferencial es el siguiente:

```
function dy=tiropar3(t,y,m,c) % version vectorizada
fac=-c*sqrt(y(1,:).^2+y(2,:).^2);
dy=zeros(size(y));
dy(1,:)=fac.*y(1,:);
dy(2,:)=fac.*y(2,:)-9.8*m;
dy(3,:)=y(1,:);
dy(4,:)=y(2,:);
```

Sobre las funciones definidas en el fichero ***tiropar3Main.m*** se pueden hacer los siguientes comentarios:

1. El programa se ha definido como ***función sin argumentos*** (línea 2) y no como fichero de comandos. La razón es para poder utilizar ***sub-funciones*** (funciones definidas en el mismo fichero), que no están permitidas en los ficheros de comandos.
2. Las condiciones iniciales (línea 7) se parecen a las de los ejemplos anteriores, pero el movimiento comienza con una ordenada negativa (por debajo del origen). Este hecho se utilizará en relación con los ***eventos***.
3. Las ecuaciones diferencias se suponen en la forma de la ecuación (14) ($\mathbf{M}(t,\mathbf{y})\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$), que permite proporcionar más información al integrador. En la línea 9 se define una matriz dispersa con "unos" en las posiciones correspondientes a los términos distintos de cero de la matriz Jacobiana del segundo miembro, esto es, a las derivadas parciales distintas de cero del vector $\mathbf{f}(t,\mathbf{y})$ respecto al vector $\mathbf{y}$. El integrador ***ode15s*** va a calcular dichas derivadas numéricamente y la información contenida en la matriz **Jp** sobre los términos no nulos le permite ahorrar mucho trabajo.
4. La estructura **options**, definida en las líneas 10-12, tiene una gran importancia, pues controla los aspectos fundamentales de la integración. Como ya se ha dicho, sus valores se establecen en la forma de parejas *parámetro/valor*. Los primeros argumentos son los valores de las tolerancias de error relativo y absoluto, ya comentados previamente. A continuación se comentan las restantes opciones. En los nombres de los parámetros MATLAB no distingue entre mayúsculas y minúsculas y no hace falta escribirlos con todas las letras: basta poner suficientes letras para evitar la ambigüedad en el nombre. Por ejemplo, ***MStateDep*** y ***mstate*** serán considerados como equivalentes a ***MStateDependence***. Es conveniente sin embargo que la elección de estos nombres no afecte a la legibilidad del código.
5. La tercera pareja de argumentos de **options** declara que la matriz de masas (parámetro ***Mass***) es definida por la función ***tiropar3Masa***, cuya referencia se da como valor. Otra posibilidad es la que se ha utilizado antes: cuando la matriz de masas es constante, se calcula previamente su valor y se incluye el nombre de la variable como valor de este argumento. La línea 11 incluye el argumento ***MStateDep***, también relacionado con la matriz de masas, que establece el tipo de dependencia de dicha matriz respecto al vector de estado **y**. Sus posibles valores son ***none***, ***weak*** y ***strong***. Otro argumento relacionado con la matriz de masas, no utilizado en este ejemplo, es ***MvPattern***, cuyo valor debe ser una matriz sparse análoga a **Jp**, utilizada para de-

finir la dependencia de los elementos de la matriz de masas respecto al vector **y** (su *sparsity pattern*).

6. El parámetro ***OutputFcn*** permite al usuario controlar la salida de resultados a medida que se van haciendo los cálculos y no solamente al final. El valor de este parámetro es una referencia de función (***@tiropar3Salida***) que será utilizada por el integrador varias veces a lo largo de la integración. MATLAB dispone de cuatro funciones de salida preprogramadas (***odeplot***, ***odephas2***, ***odephas3*** y ***odeprint***) que pueden ser utilizadas sin más que pasar como valor una referencia a ellas. En este caso se ha programado una función de salida llamada ***tiropar3Salida*** que está definida a partir de la línea 34. Esta función se llama al inicio de la integración, en cada instante de salida de resultados y al terminar la integración. El parámetro ***OutputSel*** permite definir los elementos del vector de estado en los que se está interesado. En este caso se le han pasado como valor el vector [3, 4], lo que hará que la función de salida reciba las posiciones y no las velocidades (que están en las dos primeras posiciones del vector de estado **y**).
7. La Jacobiana de la función **f**(*t*,**y**) respecto al vector **y** tiene una gran importancia, sobre todo en problemas ***stiff***. El usuario puede proporcionar una Jacobiana al integrador, pero si no lo hace éste la calcula numéricamente. El usuario puede también proporcionar una referencia de función que calcule una Jacobiana analíticamente (de modo exacto o aproximado) por medio del parámetro ***Jacobian***. Cuando no se utiliza este parámetro, el integrador calcula la Jacobiana numéricamente y también en este caso el programador puede ayudar a acelerar mucho los cálculos. El parámetro ***JPattern***, ya comentado previamente, permite indicar al integrador qué elementos de la Jacobiana son distintos de cero. Además, como una Jacobiana contiene n^2 derivadas parciales y cada derivada se calcula a partir de la diferencia entre dos evaluaciones de **f**(*t*,**y**), este cálculo puede ser muy costoso para valores grandes de ***n***. El parámetro ***JPattern*** permite reducir el cálculo de derivadas numéricas. Además, el parámetro ***Vectorized*** permite realizar este cálculo mucho más rápidamente utilizando las capacidades vectoriales de MATLAB; sus posibles valores son ***on*** y ***off***. Más adelante se verá cómo se ha vectorizado en este ejemplo la función ***tiropar3***.
8. El penúltimo parámetro que aparece en la estructura **options** es ***Events***. Este parámetro puede tener una gran importancia en simulación. En este contexto, se llaman ***eventos*** a todas aquellas circunstancias que pueden acaecer a lo largo de la simulación y que cambian su naturaleza u obligan a tomar una decisión. Por ejemplo, si se está simulando el movimiento de un vehículo todo terreno, cada vez que las ruedas pierden o vuelven a tomar contacto con el suelo se produce un cambio en el sistema a simular, pues se eliminan o añaden ecuaciones y grados de libertad. Los ***eventos*** de MATLAB permiten detectar semi-automáticamente estas situaciones y tomar las medidas adecuadas. En el ejemplo de tiro parabólico que se está considerando el único evento que se va a considerar es que el proyectil llegue al suelo, es decir, que su coordenada ***y*** se anule (se supone que el suelo es la superficie *y=0*). El valor del parámetro ***Events*** es la referencia de la función de usuario que se ocupará de gestionarlos.
9. El último parámetro de **options** es ***Stats***, que cuando está en ***on*** hace que el integrador calcule e imprima las estadísticas sobre el trabajo que ha sido necesario en la integración.
10. La línea 13 contiene la ***llamada al integrador***, en este caso a la función ***ode15s***. La línea 15 muestra –comentada, para que no se ejecute– una ***forma alternativa*** de realizar esta llamada. En el primer caso, que fue una novedad de la versión 6.0 de MATLAB, el integrador entrega todos los resultados como campos de una estructura, que en este caso se ha llamado **sol**. El listado del programa indica los significados de los campos más importantes: **sol.x** es un vector con los tiempos en los que se devuelven resultados y **sol.y** es una matriz cuyas filas son los re-

sultados correspondientes. Si están activados los eventos hay tres campos adicionales **xe**, **ye** e **ie**, que representan respectivamente los instantes de tiempo en que se han producido los eventos, los valores del vector de estado en esos instantes de tiempo, y el evento concreto (pueden controlarse varios eventos diferentes) que se ha producido en cada uno de esos instantes.

11. Tanto en la línea 13 como en la 15 aparecen detrás de **options** dos argumentos adicionales que representan la masa **m** (1) y el amortiguamiento **c** (0.001). Todos los argumentos que aparezcan detrás de **options** son siempre considerados argumentos adicionales por el integrador, que se limita a recogerlos y pasárselos a todas las funciones de usuario tales como ***tiropar3***, ***tiropar3Masa***, etc. El integrador es un mero transmisor de argumentos entre el programa principal y las restantes funciones de usuario. Es muy importante que *todas las funciones de usuario recojan estos argumentos adicionales* aunque no los necesiten, pues en otro caso se obtiene un error de *insuficiente número de argumentos*. Obsérvese que estos argumentos aparecen en las líneas 30, 34 y 54.
12. Los integradores de MATLAB pueden dar el resultado por medio de una estructura (como en la línea 13) o por medio de diversos valores de retorno (como en la línea 15 y en los ejemplos anteriores). Se utiliza una u otra forma en función del número de valores de retorno que espera recibir el usuario.
13. Cuando el resultado del integrador se recibe por medio de una estructura, los instantes en los que se proporcionan resultados no coinciden con los valores intermedios de **tspan**, sino que son ***valores elegidos por el integrador***, al igual que en el caso en que **tspan** sólo contenga el instante inicial y el final (**tspan=[tini, tend];**). Si se desean resultados en intervalos igualmente espaciados elegidos por el usuario, puede utilizarse la función ***deval*** (ver línea 25), que fue otra novedad de MATLAB 6.0. Si a esta función se le pasa la estructura solución **sol** y un vector con los instantes de tiempo en los que se desea solución, ***deval*** devuelve una matriz cuyas filas son el vector solución en dichos instantes de tiempo.
14. La línea 30 muestra el comienzo de la función ***tiropar3Masa***, que calcula la matriz de masas del sistema de ecuaciones diferenciales (14). Es muy importante que los argumentos sean los que se indica en esa línea. El argumento **t** aparece porque se supone que la matriz **M** varía con el tiempo y no con el vector de estado **y**. Si se considerara constante (como en realidad es en este caso) se podía haber pasado directamente como valor del parámetro ***Mass*** en ***odeset***. Con ***MStateDep*** igual a ***none*** se ha indicado que la matriz de masas no depende de **y**, y por eso dicho vector no se pasa como argumento. Por otra parte, como a todas las funciones de usuario, el integrador le pasa los dos argumentos adicionales **m=1** y **c=0.001** que aparecen al final de la llamada al integrador en la línea 13.
15. La función ***tiropar3Salida*** en la línea 34 tiene un tercer argumento llamado **flag** (señal, marca o bandera). A lo largo de la integración del programa esta función es llamada con tres valores diferentes de **flag**. Al principio de todo se llama con **flag** igual a ***'init'***; después, a lo largo de la integración, se llama en cada instante de salida con **flag** igual a la cadena vacía *' '*; cuando la integración ha terminado se llama con **flag** igual a ***'done'*** para que el usuario pueda hacer las últimas operaciones antes de terminar. El programador debe decidir el uso que hace de estas posibilidades. En este caso, la función ***tiropar3Salida*** lo único que hace es imprimir un mensaje por la consola y llamar a la función ***odeplot***, que es una de las funciones estándar de MATLAB. Es importante observar el código de las líneas 46-50: este código es necesario porque el integrador consigue dar pasos tan grandes que puede haber varios puntos de salida dentro de un único paso. En este caso, el argumento **t** es un vector con los diversos tiempos de salida e **y** es una matriz cuyas columnas son los vectores de estado en cada punto de salida. Si no se introduce el bucle ***for*** de la línea 47 el programa da error.

16. Llega el momento de hacer unos comentarios sobre la función que maneja los ***eventos***. El integrador necesita del usuario para detectar los eventos que se pueden producir a lo largo de la simulación. Para MATLAB un ***evento*** es siempre ***una variable cuyo valor pasa por cero***, bien en dirección ***ascendente*** o ***descendente***. Esa variable puede ser una coordenada, una distancia, una fuerza, ... Si en vez de hacerse cero tiene que alcanzar un valor determinado, bastará controlar la diferencia correspondiente. El usuario debe decir pues al integrador qué valores tiene que controlar. Esto se hace por medio de los valores de retorno de la función que gestiona les eventos, en este caso ***tiropar3Eventos*** (ver línea 54). El primer valor de retorno, **valor**, es la variable cuyo paso por cero hay que vigilar (puede ser un ***vector de valores a controlar***). El segundo valor de retorno, **esFinal**, indica si la ejecución se debe terminar o no cuando se produzca el evento, lo cual dependerá del caso estudiado y del propósito del programador. El tercer y último valor de retorno, **direccion**, indica cómo es el paso por cero, si con derivada positiva o creciendo (valor 1), o con derivada negativa o decreciendo (valor –1). La función ***tiropar3Eventos*** muestra un ejemplo de cómo se generan estos valores: el valor a controlar es la ordenada del móvil, que viene dada por **y(4)**. El evento es ***final*** cuando el móvil impacta con el suelo en su trayectoria descendente (velocidad **y(2)<0**), pero no cuando está subiendo (velocidad **y(2)>0**). De hecho, se produce un primer evento al poco tiempo de ser disparado (ya que parte de un punto con ordenada negativa), pero este evento no detiene la simulación. El segundo evento sí la para, pues se produce con trayectoria descendente.
17. Finalmente, es necesario explicar la función ***tiropar3***, que se incluye en la página 91. Dicha función está ***vectorizada***, esto es, es capaz de recibir como argumento varios vectores de estado (columnas de la matriz **y**) y calcular los correspondientes valores del vector de derivadas, que serán las columnas de la matriz **dy**. Para vectorizar esta función se ha hecho uso de los operadores (**.***) y (**.^**). Cada fila de la matriz resultado **dy** se calcula con una única instrucción, en función de las filas de la matriz **y**. Esta simple modificación (en otros ejemplos puede ser mucho más complicada) hace que los cálculos sean mucho más eficientes.

No se puede entrar con más detenimiento en estas cuestiones especializadas. Para el lector interesado en estos problemas se recomienda acudir a la ayuda de MATLAB, concretamente al tema titulado ***Initial Value Problem Solvers***, accesible desde la ventana principal del ***Help*** de MATLAB en ***Mathematics***, ***Differential Equations***, ***Initial Value Problems for ODEs and DAEs***. También puede ser muy útil consultar la información que aparece sobre ***MATLAB ODE Suite***, en la sección de ***Printable Documentation (PDF)***, en la ventana de ***Help***. Estos documentos contienen una explicación muy detallada de todas las posibilidades de las funciones referidas, así como numerosos ejemplos.

6.10.4. LAS FUNCIONES *EVAL*, *EVALC*, *FEVAL Y EVALIN*

Estas funciones tienen mucho que ver con las cadenas de caracteres, ya que necesitan la flexibilidad de éstas para alcanzar todas sus posibilidades. Las funciones para manipular cadenas de caracteres se verán en un próximo apartado.

La función ***eval('cadena de caracteres')*** hace que se evalúe como expresión de MATLAB el texto contenido entre las comillas como argumento de la función. Este texto puede ser un comando, una fórmula matemática o -en general- cualquier expresión válida de MATLAB. La función ***eval*** debe tener los valores de retorno necesarios para recoger los resultados de la expresión evaluada.

Esta forma de definir ***macros*** es particularmente útil para pasar nombres de función a otras funciones definidas en ficheros ****.m***.

El siguiente ejemplo va creando variables llamadas **A1**, **A2**, ..., **A10** utilizando la posibilidad de concatenar cadenas antes de pasárselas como argumento a la función ***eval***:

```
for n = 1:10
    eval(['A',num2str(n),' = magic(n)'])
end
```

La función ***eval()*** se puede usar también en la forma ***eval('tryString', 'catchString')***. En este caso se evalúa la cadena ***'tryString'***, y si se produce algún error se evalúa la cadena ***'catchString'***. Es una forma simplificada de gestionar errores en tiempo de ejecución.

La función ***T=evalc()*** es similar a ***eval()*** pero con la diferencia de que cualquier salida que la expresión pasada como argumento hubiera enviado a la ventana de comandos de MATLAB es capturada, y almacenada en una matriz de caracteres **T** cuyas filas terminan con el carácter '\n'.

Por su parte la función ***feval*** sirve para evaluar, dentro de dicha función, otra función cuya referencia o cuyo nombre contenido en una cadena de caracteres se le pasan como primer argumento. Es posible que este nombre se haya leído desde teclado o se haya recibido como argumento. A la función ***feval*** hay que pasarle como argumentos tanto la referencia o el nombre de la función a evaluar como sus argumentos. Por ejemplo, si dentro de una función se quiere evaluar la función ***calcular(A, b, c)***, donde el nombre ***calcular*** o la referencia ***@calcular*** se envía como argumento en la cadena ***nombre***, entonces ***feval(nombre, A, b, c)*** equivale a ***calcular(A, b, c)***.

Finalmente, la función ***evalin(ws, 'expresion')*** evalúa *'expresion'* en el espacio de trabajo ***ws***. Los posibles valores de ***ws*** son ***'caller'*** y ***'base'***, que indican el espacio de trabajo de la función que llama a ***evalin*** o el espacio de trabajo base. Los valores de retorno se recogen del modo habitual.

6.11. Distribución del esfuerzo de cálculo: *Profiler*

El ***profiler*** es una utilidad que permite saber qué tiempo de cálculo se ha gastado en cada línea de una ***función*** definida en un fichero ****.m*** o en general de un programa de MATLAB. El ***profiler*** también puede utilizarse con ***ficheros de comandos *.m***, es decir con programas que no sean funciones. Permite asimismo determinar el número de llamadas a dicha función, las funciones que la han llamado (*parent functions*), las funciones llamadas por ella (*child functions*), etc.

El ***profiler*** mejora la calidad de los programas, pues permite detectar los "cuellos de botella" de una aplicación y concentrar en ellos los esfuerzos para mejorar su eficiencia. Por ejemplo, sabiendo el número de veces que se llama a una función y el tiempo que cuesta cada llamada, se puede decidir si es mejor emplear más memoria en guardar resultados intermedios para no tener que calcular varias veces lo mismo.

El ***profiler*** ha sido mejorado en las distintas versiones de MATLAB, disponiendo de una interface de usuario propia (ver Figura 33 y siguientes). Con el ***profiler*** se puede medir el tiempo (en centésimas de segundo) empleado en cada línea del fichero, en cada llamada a una función e incluso en cada operador del lenguaje.

Para arrancar la ventana del ***profiler*** se puede ejecutar el comando ***Profiler*** en el menú ***Desktop***, utilizar el menú ***Start/Matlab*** o simplemente teclear ***profile viewer*** en la ventana de comandos. El resultado será que se abre la ventana que se muestra en la Figura 33.

Ahora se debe introducir en la ventana ***Run this code*** el nombre de la función o fichero ****.m*** que se quiere analizar y después clicar en el botón ***Start Profiling***. Inmediatamente el programa empieza a ejecutarse bajo la supervisión del ***profiler***, lo cual se sabe porque en la parte superior derecha de la ventana aparece una marca verde y comienza a correr el tiempo. Al acabar la ejecución se muestra el resumen de resultados (***Profile Summary***), que para el programa ***tiroparMain4.m*** explicado en el apartado 6.10.3, es el mostrado en la Figura 34. En esta figura se muestra un informe del nº de veces que ha sido llamada cada función y del tiempo total que se ha empleado en ella.

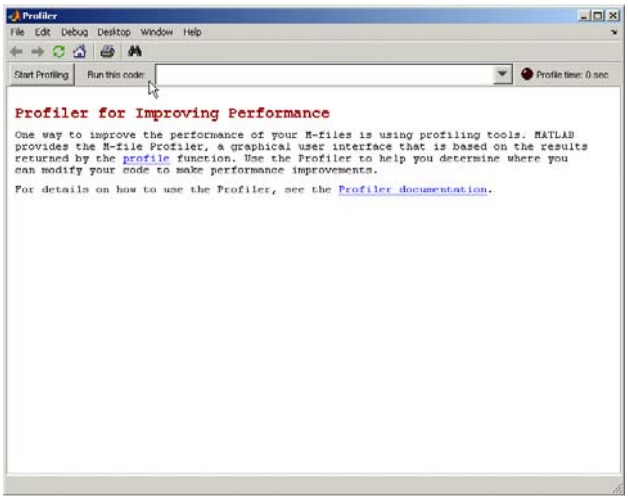


Figura 33. Ventana inicial del ***Profiler***.

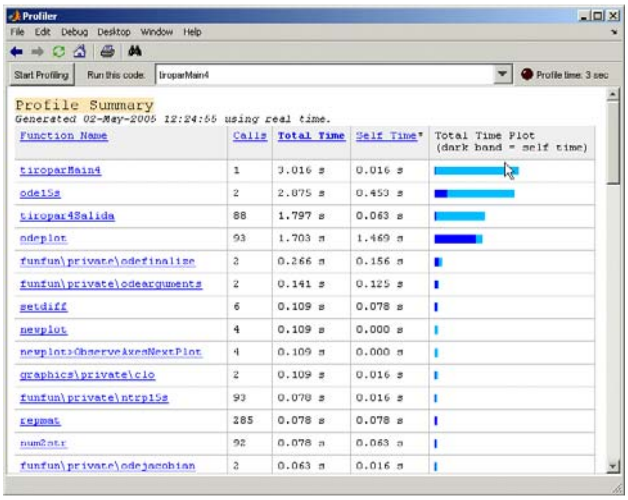


Figura 34. ***Profile Summary*** para ***tiroparMain4***.

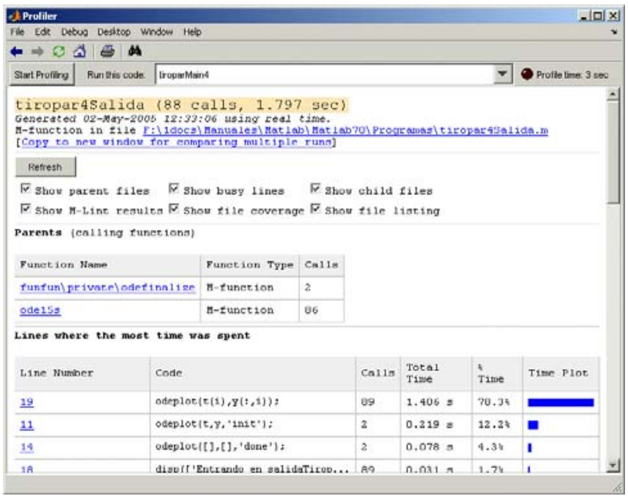


Figura 35. Información sobre la función ***tiropar4salida***.

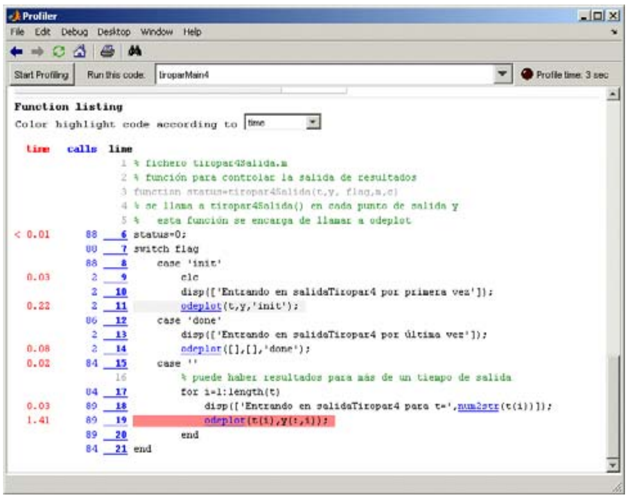


Figura 36. Información adicional sobre ***tiropar4salida***.

Clicando sobre el enlace a la función ***tiropar4salida*** se muestra la información de la Figura 35 y de la Figura 36. En la parte de arriba de la ventana (Figura 35) se muestra el número de veces que ha sido llamada cada línea de código y el tiempo empleado en ella. En la parte inferior de la ventana (Figura 36) aparece una información similar referida al código fuente del fichero. Las sentencias que se han llevado más tiempo de CPU aparecen coloreadas en un tono rosa de intensidad creciente.

Una información particularmente interesante es la proporcionada bajo el epígrafe ***M-Lint results***, (no mostrados en las figuras) que contiene sugerencias para mejorar la eficiencia del programa. Por ejemplo, se avisa de la existencia de variables que se calculan pero no se utilizan posteriormente, de la existencia de alternativas más eficientes, etc.

El ***profiler*** proporciona también información sobre la función padre (la que la ha llamado) y las funciones hijas de cada función.

Otra forma de llamar al ***profiler*** es por medio de la función ***profile***, que se intercala en el código fuente en la forma (se supone que estas líneas forman parte de un fichero ****.m***):

```
profile on -detail 'builtin';
sol=ode15s(@tiropar4,tspan2,y0,options,1,0.001);
profile viewer;
```

donde la primera línea activa el ***profiler*** a la vez que define el ***grado de detalle*** que se desea y el tipo de tiempo que se desea medir. La segunda línea es una llamada a la función ***ode15s*** que a su vez llama a muchas otras funciones y la tercera línea detiene el ***profiler*** y le pide un informe con los resultados calculados. Los informes presentados por el Pprofiler son similares a los de la Figura 34 y siguientes, aunque en este caso no se muestra un informe total sobre la función ***tiroparMain4***, sino sólo sobre la parte de código que está entre ***profile on*** y ***profile viewer***.

Existen dos posibles grados de detalle respecto a la información que se le pide al ***profiler***:

'mmex'	determina el tiempo utilizado por funciones y sub-funciones definidas en ficheros ****.m*** y ****.mex***. Ésta es la opción por defecto.
'builtin'	como el anterior incluyendo las ***funciones intrínsecas*** de MATLAB.

Otros posibles comandos relacionados con el ***profiler*** de MATLAB son los siguientes:

profile viewer	abre la ventana del ***profiler*** mostrada en la Figura 34.
profile on	activa el ***profiler*** poniendo a cero los contadores
profile on –detail level	como el anterior, pero con el grado de detalle indicado
profile on –history	activa el ***profiler*** con información sobre el orden de las llamadas a las funciones (puede registrar hasta 10000 llamadas)
profile off	desactiva el ***profiler*** sin poner a cero los contadores
profile resume	vuelve a activar el ***profiler*** sin poner a cero los contadores
profile clear	pone a cero los contadores
s = profile('status')	muestra una estructura conteniendo los datos del ***profile***
stats = profile('info')	detiene el ***profiler*** y muestra una estructura con los resultados (consultar el ***Help*** para más información sobre las variables miembro de estas estructuras)

El ***profiler*** se puede aplicar a funciones y a ficheros de comandos. La ejecución de un programa con el ***profiler*** puede ser bastante más lenta que sin él, por el trabajo extra que conlleva.

7. INTERFACES DE MATLAB CON OTROS LENGUAJES

7.1. Interfaces de MATLAB con DLLs genéricas

7.1.1. INTRODUCCIÓN

Una librería compartida es una colección de funciones ejecutables listas para ser utilizadas en una o más aplicaciones. En este sentido, MATLAB permite utilizar librerías externas que se hayan generado en sistemas MS-Windows y Linux.

Los ficheros fuente de la librería se precompilan y ensamblan, y de este modo se obtiene un fichero con la extensión ".dll" (dynamic link library) en MS Windows o ".so" (shared object) en UNIX y Linux. En tiempo de ejecución de la aplicación que las va a utilizar las funciones de la librería son cargadas en memoria y ejecutadas.

MATLAB permite usar las funciones de estas librerías programadas en C. El acceso a las funciones se realiza a través de una interface de línea de comandos. Esta interface ofrece la posibilidad de cargar una librería externa en MATLAB y acceder a cualquiera de las funciones definidas en dicha librería. Aunque los tipos de datos son diferentes en MATLAB y en C, en muchos casos es posible pasar los tipos de MATLAB a C sin tenerse que preocupar de la conversión de datos, porque MATLAB la realiza de forma automática.

Esta interface permite también usar funciones programadas en otros lenguajes distintos de C siempre que dichas funciones tengan una interface con C. Por ejemplo, es posible llamar a una DLL programada en Visual Basic si existe un fichero de declaraciones C para dicha librería.

7.1.2. CARGAR Y LIBERAR LAS LIBRERÍAS DE MEMORIA

Para que MATLAB tenga la posibilidad de acceder a una función externa de una librería es necesario en primer lugar cargar la librería en memoria. Una vez cargada, ya se puede llamar a cualquiera de sus funciones. Cuando la librería ya no se necesita es conveniente borrarla para liberar memoria.

Para cargar una librería en MATLAB se usa la función ***loadlibrary***. La sintáxis de la función es la siguiente:

```
>> loadlibrary('shrlib', 'hfile')
```

donde ***shrlib*** es el nombre de la librería y ***hfile*** es el nombre del fichero que contiene la declaración de las funciones (fichero de encabezamiento o ***header***).

Como ejemplo se va a cargar en memoria la librería de MATLAB ***libmx*** que contiene las funciones ***mx*** (funciones en C que permiten trabajar con los ***mxArrays***, es decir, con cualquiera de las variables de MATLAB). En la primera sentencia mostrada a continuación se guarda en la variable ***hfile*** la dirección del fichero ***matrix.h*** que contiene la declaración de las funciones ***mx***. La segunda sentencia carga en memoria la librería ***libmx.dll***.

```
>> hfile = [matlabroot '\extern\include\matrix.h'];
>> loadlibrary('libmx', hfile)
```

Si ya no se necesitan estas funciones se puede liberar la memoria mediante la función ***unloadlibrary***, en la forma:

```
>> unloadlibrary libmx
```

La función ***libisloaded*** devuelve un 1 (true) si la función esta cargada y 0 (false) si no lo está. Esta función se podría utilizar en la forma siguiente:

```
if libisloaded('libmx')==1 ...
```

7.1.3. CONSEGUIR INFORMACIÓN ACERCA DE LA LIBRERÍA

Para saber qué funciones tiene disponibles una librería cargada en memoria se pueden usar las dos siguientes funciones:

```
>> libfunctions('libname')
>> libfunctionsview('libname')
```

La principal diferencia es que ***libfunctions*** muestra la información en la ventana de comandos de MATLAB y ***libfunctionsview*** muestra la información en una ventana nueva.

Si se usa ***libfunctions*** con el nombre de la librería como único argumento MATLAB devuelve los nombres de las funciones de la librería sin especificar sus argumentos. Éstos son mostrados si se utiliza la opción "***–full***". Los tipos de datos que acompañan a los argumentos son los tipos de MATLAB correspondientes a los tipos de C. Como ejemplo obsérvese la siguiente respuesta de MATLAB:

```
>> libfunctions libmx –full
Functions in library mx:
[int32, MATLAB array, cstring] mxAddField(MATLAB array, cstring)
[cstring, MATLAB array] mxArrayToString(MATLAB array)
```

Además, la función ***libfunctionsview*** crea una nueva ventana donde muestra toda la información de las funciones disponibles de la librería.

7.1.4. LLAMADA A LAS FUNCIONES DE UNA LIBRERÍA

La función ***calllib*** sirve para llamar a las funciones de la librería. Se debe especificar el nombre de la librería, el nombre de la función y los argumentos que hay que pasar a dicha función:

```
>> calllib('libname', 'funcname', arg1, ..., argN)
```

El siguiente ejemplo llama a funciones de la librería ***libmx***:

```
>> y = rand(4, 7, 2);
>> calllib('libmx', 'mxGetNumberOfElements', y)
ans = 56
>> calllib('libmx', 'mxGetClassID', y)
ans = mxDOUBLE_CLASS
```

En el paso de argumentos se deben tener en cuenta las siguientes reglas generales:

– Algunos tipos de argumentos muy utilizados en C, tales como punteros o estructuras predefinidas, son muy diferentes a los tipos de datos estándar de MATLAB. En estos casos existe la opción de pasar un tipo de dato estándar de MATLAB y arriesgarse a que MATLAB haga la conversión, o convertir los datos explícitamente con funciones tales como ***libstruct*** y ***libpointer***, que se explicarán más adelante.

– En C muchos argumentos de entrada se pasan normalmente ***por referencia***. Aunque MATLAB no soporta el paso de argumentos por referencia, es posible en MATLAB crear argumentos compatibles con las referencias de C.

– Las funciones de C suelen devolver resultados por medio de argumentos pasados por referencia. En este caso, MATLAB crea valores de retorno adicionales para devolver estos valores.

7.1.5. CONVERSIÓN DE DATOS

La Tabla 2 y la Tabla 3 muestran la equivalencia de tipos de datos entre C y MATLAB.

Tipo de C (en un ordenador de 32 bit)	**Tipo equivalente de MATLAB**
char, byte	int8
unsigned char, byte	uint8
short	int16
unsigned short	uint16
int, long	int32
unsigned int, unsigned long	uint32
float	single
double	double
char *	cstring (1xn char array)
*char[]	cell array of strings

Tabla 2. Tipos de datos primitivos de MATLAB.

Tipo de C (en un ordenador de 32 bit)	**Tipo equivalente de MATLAB**
int *	intPtr
*char	stringPtr
**char	stringPtrPtr
float *	singlePtr
double *	doublePtr
mxArray *	Matlab array
void *	voidPtr
void **	voidPtrPtr
type **	*type*PtrPtr

Tabla 3. Tipos de datos extendidos de MATLAB

Para los tipos primitivos, MATLAB automáticamente convierte cualquier argumento de MATLAB al tipo de dato C esperado. Por ejemplo se puede pasar un ***double*** a una función que espera recibir un ***integer*** y MATLAB realiza la conversión. MATLAB también convierte un argumento pasado por valor a un argumento pasado por referencia, cuando la función externa requiere que el argumento sea pasado de esta forma.

Por ejemplo la siguiente función C, que permuta dos argumentos pasados por referencia, se ha compilado en una librería llamada ***swapLibrary***:

```
void swap(double *x, double *y) {
    double aux = *x;
    *x = *y;
    *y = aux;
    return;
}
```

Se puede llamar a la función de la siguiente manera:

```
a = 1.78; b = 5.42;
calllib('swapLibrary', 'swap', a, b);
```

Para argumentos del tipo ***char**** se puede pasar un string de MATLAB.

7.1.6. PASO DE ESTRUCTURAS COMO ARGUMENTOS

Para pasar estructuras como argumentos a funciones de C hay que crear las estructuras equivalentes en MATLAB. Para ello hay que conocer los nombres de los campos y el tipo de dato de cada campo. Si estos datos son conocidos, bien porque se posee la documentación de la librería, bien porque

se puede encontrar la definición de la estructura en el fichero de declaraciones de la librería, se puede construir directamente la estructura equivalente en MATLAB poniendo los mismos nombres a los campos.

Si no se conoce el nombre y el tipo de los campos existe otra posibilidad. Por ejemplo, al considerar las funciones disponibles en una librería se observa la siguiente declaración:

```
double addStructFields(c_struct)
```

En este caso se puede emplear la función ***libstruct*** de MATLAB, que devuelve un objeto vacio con una estructura equivalente a la de la estructura de C.

```
>> s = libstruct('c_struct')
```

Ahora, con la función ***get*** se puede averiguar el nombre de los campos:

```
>> get(s)
    p1: 0
    p2: 0
    p3: 0
```

A continuación se pueden dar los valores deseados a los campos y llamar a la función:

```
>> s.p1 = 476;  s.p2 = -299;  s.p3 = 1000;
>> calllib('shrlibsample', 'addStructFields', s);
```

7.1.7. PASO DE ARGUMENTOS POR REFERENCIA

Como se ha comentado anteriormente, MATLAB se encarga de comprobar si los argumentos se han pasado correctamente y es capaz de convertir un argumento por valor a otro por referencia cuando es necesario.

En ciertas ocasiones es conveniente utilizar un tipo de dato de MATLAB equivalente a las referencias de C, como por ejemplo:

- Si se desean modificar los datos en los argumentos de entrada.
- Si se está pasando una gran cantidad de datos a la función y no se desea que MATLAB haga copias de dichos datos (MATLAB sólo saca copias de los argumentos de entrada si son modificados dentro de la función).
- Cuando un argumento de una función tiene más de un nivel de referencia (puntero a puntero), por ejemplo ***double *****, no es conveniente dejar que Matlab haga la conversión.

En estos casos se puede usar la función ***libpointer*** de MATLAB, que sirve para construir punteros a diferentes tipos de datos. Su sintáxis es la siguiente:

```
>> p = libpointer('type', 'value')
```

Esta función se entiende mejor por medio de un ejemplo. A continuación se muestra una función en C que multiplica por cinco una variable que se ha recibido como argumento pasado por referencia:

```
double *multDoubleRef(double *x) {
    *x *=5;
    return x;
}
```

En MATLAB se puede definir la variable, construir una referencia y verificar su contenido:

```
>> x = 15;
>> xp = libpointer('doublePtr', x);
>> get(xp)
    Value:  15
    DataType:  'doublePtr'
```

Finalmente se llama a la función y se comprueba el resultado:

```
>> calllib('shrlibsample', 'multDoubleRef', xp);
>> get(xp, 'Value')
    ans = 75
```

7.2. Llamar desde MATLAB funciones programadas en C o Fortran

7.2.1. INTRODUCCIÓN A LOS FICHEROS MEX

Es posible llamar desde MATLAB a funciones programadas en C y en Fortran como si fueran funciones propias de MATLAB. De este modo, una función *.m de MATLAB puede ser sustituida por una función programada en C o en Fortran que se llama exactamente en la misma forma. Para que esto sea posible las funciones programadas en C y Fortran han de cumplir una serie de requisitos que se explican más adelante. Estas funciones se compilan y se generan librerías compartidas que son las denominadas funciones MEX. Las funciones MEX son funciones ejecutables "*.dll" ó "*.so" que pueden ser cargadas y ejecutadas por MATLAB de forma automática.

Las funciones MEX tienen varias aplicaciones:

- Evitan tener que reescribir en MATLAB funciones que ya han sido escritas en C o Fortran.
- Por motivos de eficiencia puede ser interesante reescribir en C o Fortran las funciones críticas o que consumen más CPU del programa.

Las funciones MEX tienen una extensión diferente en función de los sistemas operativos en que hayan sido generadas. En la siguiente tabla se puede ver la extensión que corresponde a cada sistema operativo.

Sistema operativo	Extensión del fichero MEX
Sun Solaris	.mexsol
HP-UX	.mexhpux
Linux	.mexglx
MacIntosh	.mexmac
Windows	.dll (hasta Matlab 7.0)
	.mexw32 (desde Matlab7.1)

Tabla 4. Extensiones de los ficheros MEX.

Conviene insistir en que se puede llamar un fichero MEX exactamente de la misma forma que se llama a un fichero *.m de MATLAB. Además, hay que tener en cuenta que si al buscar en el *path* se encuentran en el mismo directorio un fichero MEX y un fichero *.m con el mismo nombre, el fichero MEX tiene precedencia y es el que se ejecuta.

7.2.2. CONSTRUCCIÓN DE FICHEROS MEX

La primera vez que se crea un fichero MEX en un ordenador, hay que comenzar configurando el compilador que se va a utilizar. Esto se realiza tecleando en la consola de MATLAB:

```
>> mex -setup
```

MATLAB responde de la forma siguiente:

```
Please choose your compiler for building external interface (MEX) files:
Would you like mex to locate installed compilers [y]/n?
```

Si se contesta afirmativamente, MATLAB localiza los compiladores instalados en el ordenador. Después pide que se seleccione el compilador que se desea utilizar:

```
Select a compiler:
```

```
[1] Compaq Visual Fortran version 6.6 in C:\ARCHIVOS DE PROGRAMA\MICROSOFT VISUAL STUDIO
[2] Lcc C version 2.4 in C:\MATLAB7\sys\lcc
[3] Microsoft Visual C/C++ version 7.0 in C:\Archivos de programa\Microsoft Visual Studio .NET
[4] Microsoft Visual C/C++ version 6.0 in C:\Archivos de programa\Microsoft Visual Studio
[0] None
Compiler:
```

Una opción que siempre va a funcionar es seleccionar el compilador de C que incorpora MATLAB (**Lcc**, es decir, la opción 2). MATLAB pide que se confirme la elección y a continuación configura el compilador:

```
Compiler: 2
Please verify your choices:
Compiler: Lcc C 2.4
Location: C:\MATLAB7\sys\lcc
Are these correct?([y]/n): y
Try to update options file: MathWorks\MATLAB\R14\mexopts.bat
From template: C:\MATLAB7\BIN\WIN32\mexopts\lccopts.bat
Done . . .
```

NOTA: Conviene tener en cuenta que hasta la versión 7.1 de MATLAB la extensión de los ficheros MEX era ".dll". A partir de la versión 7.1 es ".mexw32". Esto quiere decir que los ficheros MEX generados por MATLAB 7.1 no podrán ser ejecutados por versiones anteriores. Para mantener la compatibilidad se debe utilizar la opción -output con la extensión ".dll" en el nombre del fichero.

La sintaxis del comando para compilar y crear un fichero MEX a partir de lo que se va a llamar un fichero C-MEX (un fichero C que cumple las condiciones necesarias para poder crear con él un fichero MEX) es la siguiente:

```
>> mex filename.c –output filename.dll
```

El resultado de esta operación es un fichero MEX que en MS Windows se llama filename.dll y que se almacena en el directorio actual.

Como es fácil de imaginar hay diversas opciones que permiten modificar las etapas de compilación y linkado de las funciones. Las opciones principales son mostradas en la Tabla 5.

Opción	**Función**
-c	Sólo compila no linka
-g	El ejecutable incluye información para depurar la función
-O	Se optimiza el ejecutable
-outdir <name>	Se indica el directorio donde se guarda la función
-output <name>	Permite cambiar el nombre del fichero mex
- v	Se saca por pantalla cada paso del compilador

Tabla 5. Opciones de la utilidad mex.

7.2.3. CREACIÓN DE FICHEROS MEX EN C

El código fuente de un fichero MEX programado en C tiene dos partes. La primera parte contiene el código de la función C que se quiere implementar como fichero MEX. La segunda parte es la función ***mexFunction*** que hace de interface entre C y MATLAB.

La función ***mexFunction*** tiene cuatro argumentos: **prhs**, **nrhs**, **plhs** y **nlhs**. Estos argumentos tienen los siguientes significados:

1. **prhs** es un vector de punteros a los valores de los argumentos de entrada (*right hand side arguments*) que se van a pasar a la función C.
2. **nhrs** es el número de argumentos de entrada de la función.

3. **plhs** y **nlhs** son análogos pero referidos a los argumentos de salida (*left hand side arguments*).

Antes de seguir adelante con esta explicación es conveniente decir algo sobre los ***mxArrays***, que los únicos objetos con los que trabaja MATLAB. Todos los tipos de variables de MATLAB (escalares, vectores, matrices, cadenas de caracteres, estructuras, vectores de celdas, etc.) son ***mxArrays***. Para cada ***mxArray*** MATLAB almacena el tipo, las dimensiones, los datos, si es real o complejo (para datos numéricos), el número de campos y sus nombres para las estructuras, etc. MATLAB dispone de un gran número de funciones C para trabajar con ***mxArrays***, que pueden encontrarse buscando "MX Array Manipulation (C)" en el ***Help***. Algunas de estas funciones se deben utilizar para construir la ***mexFunction***, tal como se explica a continuación.

7.2.4. Ejemplo de función MEX programada en C

Una función de MATLAB particularmente ineficiente es la función ***cross***, que calcula el producto vectorial de dos vectores. Dicha función se utiliza en la forma:

```
>> c = cross(a,b);
```

Los factores **a** y **b** deben ser vectores de dimensión 3, y también el resultado **c** es un vector de dimensión 3. Da igual que sean vectores fila o columna.

El siguiente programa compara tres funciones para calcular el producto vectorial de vectores: la función ***cross*** de MATLAB, la función ***prodVect.m*** y la función ***prodVectC.c***. La función ***prodVect.m*** se ha programado con MATLAB de la forma siguiente:

```
function c=prodVect(a,b)
c=zeros(3,1);
c(1)=a(2)*b(3)-a(3)*b(2);
c(2)=a(3)*b(1)-a(1)*b(3);
c(3)=a(1)*b(2)-a(2)*b(1);
```

Por su parte, la función ***prodVectC.c*** es una función C que se va a ejecutar como fichero MEX. A continuación se muestra el contenido del fichero ***prodVectC.c***, que contiene tanto la función que calcula el producto vectorial como la ***mexFunction*** de interface entre C y MATLAB:

```
// fichero prodVectC.c
#include "mex.h"

void prodVectC(const double* a, const double* b, double* c) {
    c[0] = a[1]*b[2]-a[2]*b[1];
    c[1] = a[2]*b[0]-a[0]*b[2];
    c[2] = a[0]*b[1]-a[1]*b[0];
}

// This function will be called by Matlab in the form    c = prodVectC(a,b);
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    double *a, *b;   // inputs
    double *c;       // outputs
    // Check for proper number of arguments
    if(nrhs != 2) {
        mexErrMsgTxt("Error in number of inputs.");
    } else if (nlhs != 1) {
        mexErrMsgTxt("Error in number of outputs");
    }
    // Create matrix for the return argument
    plhs[0] = mxCreateDoubleMatrix(3, 1, mxREAL);
    // Assign pointers to each input and output
    a = mxGetPr(prhs[0]);   b = mxGetPr(prhs[1]);
    c = mxGetPr(plhs[0]);
    // Call the function written in C
    prodVectC(a,b,c);
}
```

El siguiente programa de MATLAB utiliza las funciones ***tic*** y ***toc*** para medir los tiempos con las tres formas de realizar el producto vectorial:

```
% fichero pruebaMEX.m
clear all
n=50000;

% Se determina la eficiencia del producto vectorial de Matlab
tic
s=0;
for i=1:n
    a=rand(3,1); b=rand(3,1);
    c=cross(a,b);
    s=s+norm(c);
end
toc

% ahora con un programa *.m propio
tic
s=0;
for i=1:n
    a=rand(3,1); b=rand(3,1);
    c=prodVect(a,b);
    s=s+norm(c);
end
toc

% ahora con un programa propio en C
tic
s=0;
for i=1:n
    a=rand(3,1); b=rand(3,1);
    c=prodVectC(a,b);
    s=s+norm(c);
end
toc
disp('Ya he terminado')
```

Los resultados que se obtienen en la consola de MATLAB indican que la función en C es la más eficiente de las tres (aunque éste es un caso particularmente favorable para los compiladores just-in-time de los ficheros *.m de MATLAB):

```
>> pruebaMEX
Elapsed time is 5.598874 seconds.
Elapsed time is 1.261900 seconds.
Elapsed time is 0.887502 seconds.
Ya he terminado
```

A continuación se describe con más generalidad la creación de ficheros MEX. En la Figura 37, inspirada en el ***Help*** de MATLAB, se ha querido mostrar cómo se realiza la comunicación entre C y MATLAB, cómo llegan los datos al fichero MEX, qué es lo que se hace con estos datos en ***mexFunction*** y cómo se devuelven finalmente los resultados a MATLAB.

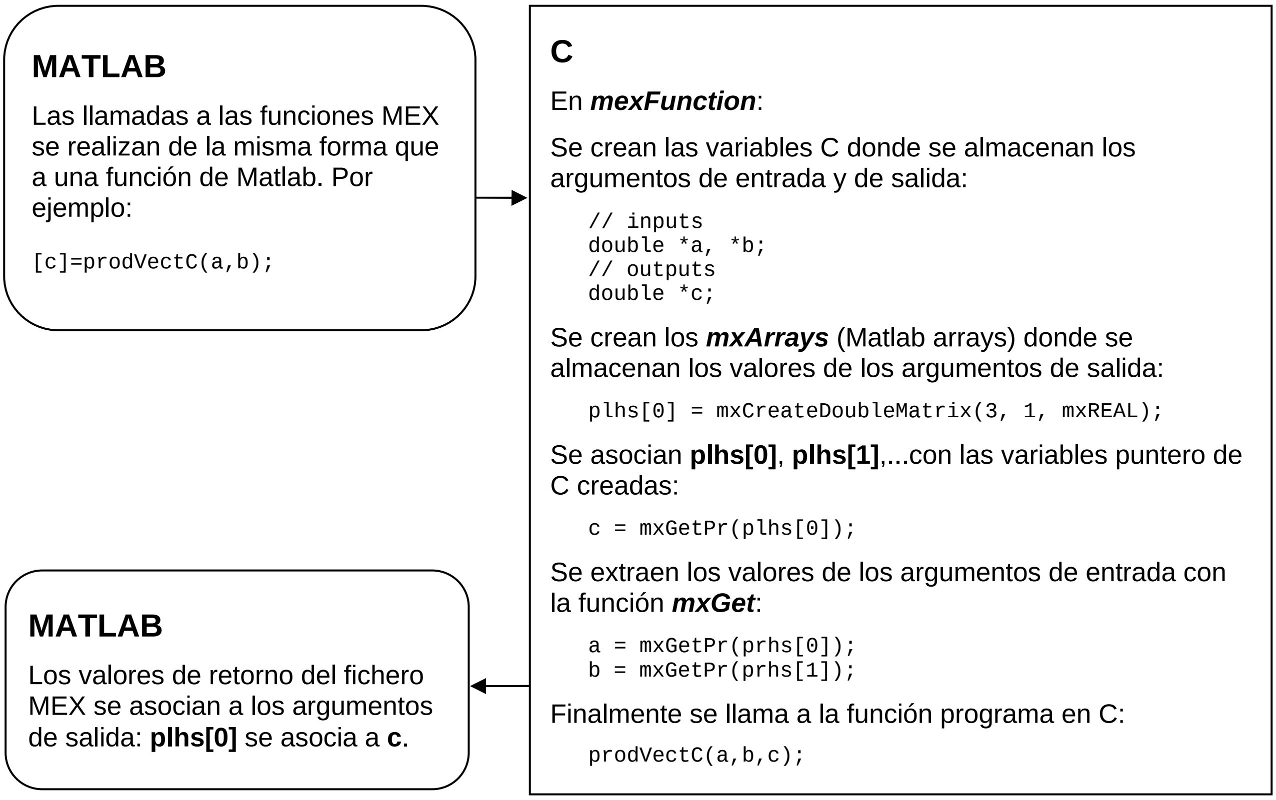


Figura 37. Esquema general de creación de una función MEX.

Los ficheros MEX deben incluir la librería "mex.h" donde está declarada la función ***mexFunction***, cuya cabecera es la siguiente:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

Como ya se ha dicho, los argumentos **plhs** y **prhs** son vectores de punteros a los argumentos de entrada (*right hand side*) y de salida (*left hand side*) del fichero MEX. Hay que señalar que ambos están declarados de tipo ***mxArray****, es decir que son variables de MATLAB. En el ejemplo de la función ***prodVectC***, **prhs** es un array de dos elementos con dos punteros a dos variables ***mxArray*** llamadas ***a*** y ***b***, y **plhs** es un array de un elemento con un puntero NULL. Este puntero es NULL porque los argumentos de salida no se crean hasta que se ejecuta la función C, que en este caso es la función ***prodVectC***. Por este motivo siempre hay que crear arrays para los valores de retorno o salidas y asignarlos a los componentes de **plhs**.

7.2.5. Depurar ficheros MEX en C en Windows

Es interesante poder depurar funciones de cualquier tipo, para poder comprobar que funcionan tal como se había previsto. Para ello, al compilar se debe incluir la información para el debugger. Esto se hace con la opción -g.

```
mex -g filename.c
```

Se va a explicar para el caso de que se use un compilador de Microsoft. En caso de que se utilicen otros compiladores se recomienda buscar la información en la documentación de MATLAB.

Para depurar primero se abre una ventana de comandos de DOS y se escribe:

```
msdev filename.dll
```

Después en el entorno de Microsoft se entra en el menu **Project**, se selecciona **Settings** y dentro de **Settings** en **Debug**. Por último en el campo **Executable for debug session** hay que poner la dirección de MATLAB.

Una vez que se han completado los pasos anteriores se puede abrir el código fuente de la función y poner break points en las líneas de código que se desee. Se lanza dentro del entorno de Visual Studio el debug. Ahora es posible ejecutar el fichero MEX en MATLAB y usar el entorno de depuración de Microsoft.

7.2.6. Depurar ficheros MEX en C en UNIX

Es necesario arrancar MATLAB desde el ***debugger***. Para ello hay que especificar el nombre del debugger que se va a usar con la opción -D al arrancar MATLAB.

Este ejemplo muestra como depurar la función ***yprime.c*** en Solaris usando el debugger de UNIX dbx.

```
unix> mex -g yprime.c
unix> matlab -Ddbx
<dbx> stop dlopen <matlab>/extern/examples/mex/yprime.mexsol
```

Una vez que el debugger carga MATLAB en memoria se puede empezar a usar utilizando el comando ***run***:

```
<dbx> run
```

Ahora se lanza el fichero MEX que se desea depurar. Antes de ejecutar el fichero MEX el programa devuelve el control al debugger.

```
>> yprime(1,1:4)
<dbx> stop in 'yprime.mexsol' mexFunction
```

Puede que sea necesario informar al debugger de donde se ha cargado el fichero MEX o el nombre del fichero, en cualquier caso MATLAB solicitará la información que necesite. En este momento se esta listo para comenzar a depurar. Se puede editar el código fuente del fichero MEX y poner breakpoints. Es conveniente poner un breakpoint al comienzo de la ***mexFunction***. Para continuar después de un breakpoint se emplea el siguiente comando.

```
<dbx> cont
```

Cuando el debugger esta parado en un breakpoint es posible utilizar todas las capacidades del debugger para examinar variables, mostrar las posiciones de memoria a inspeccionar el valor de los registros. Es conveniente estudiar las posibilidades que ofrece el debugger en su documentación.

8. GRÁFICOS BIDIMENSIONALES

A estas alturas, después de ver cómo funciona este programa, a nadie le puede resultar extraño que los gráficos 2-D de MATLAB estén fundamentalmente orientados a la representación gráfica de vectores (y matrices). En el caso más sencillo los argumentos básicos de la función ***plot*** van a ser vectores. Cuando una matriz aparezca como argumento, se considerará como un conjunto de vectores columna (en algunos casos también de vectores fila).

MATLAB utiliza un tipo especial de ventanas para realizar las operaciones gráficas. Ciertos comandos abren una ventana nueva y otros dibujan sobre la ventana activa, bien sustituyendo lo que hubiera en ella, bien añadiendo nuevos elementos gráficos a un dibujo anterior. Todo esto se verá con más detalle en las siguientes secciones.

8.1. Funciones gráficas 2D elementales

MATLAB dispone de cinco funciones básicas para crear gráficos 2-D. Estas funciones se diferencian principalmente por el *tipo de escala* que utilizan en los ejes de abscisas y de ordenadas. Estas cuatro funciones son las siguientes:

plot()	crea un gráfico a partir de vectores y/o columnas de matrices, con escalas lineales sobre ambos ejes
plotyy()	dibuja dos funciones con dos escalas diferentes para las ordenadas, una a la derecha y otra a la izquierda de la figura.
loglog()	ídem con escala logarítmica en ambos ejes
semilogx()	ídem con escala lineal en el eje de ordenadas y logarítmica en el eje de abscisas
semilogy()	ídem con escala lineal en el eje de abscisas y logarítmica en el eje de ordenadas

En lo sucesivo se hará referencia casi exclusiva a la primera de estas funciones (***plot***). Las demás se pueden utilizar de un modo similar.

Existen además otras funciones orientadas a añadir títulos al gráfico, a cada uno de los ejes, a dibujar una cuadrícula auxiliar, a introducir texto, etc. Estas funciones son las siguientes:

title('título')	añade un título al dibujo
xlabel('tal')	añade una etiqueta al eje de abscisas. Con ***xlabel off*** desaparece
ylabel('cual')	añade una etiqueta al eje de ordenadas. Con ***ylabel off*** desaparece
text(x,y,'texto')	introduce 'texto' en el lugar especificado por las coordenadas **x** e **y**. Si **x** e **y** son vectores, el texto se repite por cada par de elementos. Si **texto** es también un vector de cadenas de texto de la misma dimensión, cada elemento se escribe en las coordenadas correspondientes
gtext('texto')	introduce **texto** con ayuda del ratón: el cursor cambia de forma y se espera un clic para introducir el texto en esa posición
legend()	define rótulos para las distintas líneas o ejes utilizados en la figura. Para más detalle, consultar el ***Help***
grid	activa la inclusión de una cuadrícula en el dibujo. Con ***grid off*** desaparece la cuadrícula

Borrar texto (u otros elementos gráficos) es un poco más complicado; de hecho, hay que preverlo de antemano. Para poder hacerlo hay que recuperar previamente el *valor de retorno* del comando con el cual se ha creado. Después hay que llamar a la función ***delete*** con ese valor como argumento. Considérese el siguiente ejemplo:

```
>> v = text(1,.0,'seno')
v =
   76.0001
```

```
>> delete(v)
```

Los dos grupos de funciones anteriores no actúan de la misma forma. Así, la función ***plot*** dibuja una nueva figura en la ventana activa (en todo momento MATLAB tiene una ventana activa de entre todos las ventanas gráficas abiertas), o abre una nueva figura si no hay ninguna abierta, sustituyendo cualquier cosa que hubiera dibujada anteriormente en esa ventana. Para verlo, se comenzará creando un par de vectores **x** e **y** con los que trabajar:

```
>> x=[-10:0.2:10]; y=sin(x);
```

Ahora se deben ejecutar los comandos siguientes (se comienza cerrando la ventana activa, para que al crear la nueva ventana aparezca en primer plano):

```
>> close         % se cierra la ventana gráfica activa anterior
>> grid          % se crea una ventana con una cuadrícula
>> plot(x,y)     % se dibuja la función seno borrando la cuadrícula
```

Se puede observar la diferencia con la secuencia que sigue:

```
>> close
>> plot(x,y)     % se crea una ventana y se dibuja la función seno
>> grid          % se añade la cuadrícula sin borrar la función seno
```

En el primer caso MATLAB ha creado la cuadrícula en una ventana nueva y luego la ha borrado al ejecutar la función ***plot***. En el segundo caso, primero ha dibujado la función y luego ha añadido la cuadrícula. Esto es así porque hay funciones como ***plot*** que por defecto crean una nueva figura, y otras funciones como ***grid*** que se aplican a la ventana activa modificándola, y sólo crean una ventana nueva cuando no existe ninguna ya creada. Más adelante se verá que con la función ***hold*** pueden añadirse gráficos a una figura ya existente respetando su contenido.

8.1.1. FUNCIÓN *PLOT*

Esta es la función clave de todos los gráficos 2-D en MATLAB. Ya se ha dicho que el elemento básico de los gráficos bidimensionales es el **vector**. Se utilizan también cadenas de 1, 2 ó 3 caracteres para indicar *colores* y *tipos de línea*. La función ***plot()***, en sus diversas variantes, no hace otra cosa que dibujar vectores. Un ejemplo muy sencillo de esta función, en el que se le pasa un único vector como argumento, es el siguiente:

```
>> x=[1 3 2 4 5 3]
x =
     1     3     2     4     5     3
>> plot(x)
```

El resultado de este comando es que se abre una ventana mostrando el gráfico de la Figura 38. Por defecto, los distintos puntos del gráfico se unen con una línea continua. También por defecto, el color que se utiliza para la primera línea es el azul.

Cuando a la función ***plot()*** se le pasa un único vector –real– como argumento, dicha función dibuja en ordenadas el valor de los **n** elementos del vector frente a los índices 1, 2, ... **n** del mismo en abscisas. Más adelante se verá que si el vector es complejo, el funcionamiento es bastante diferente.

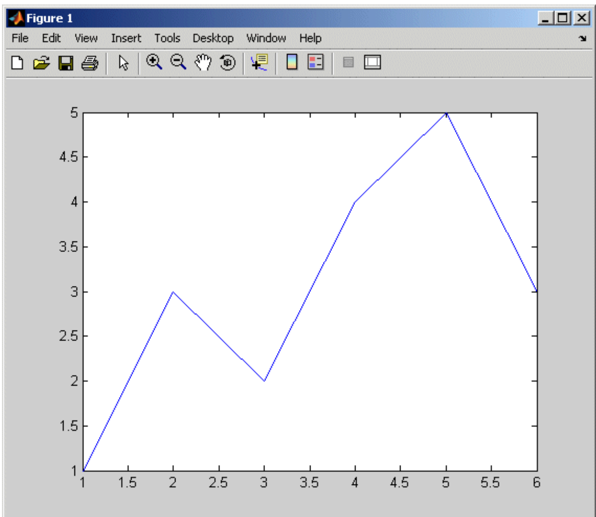


Figura 38. Gráfico del vector x=[1 3 2 4 5 3].

En la pantalla de su ordenador se habrá visto que MATLAB utiliza por defecto color blanco para el fondo de la pantalla y otros colores más oscuros para los ejes y las gráficas.

Una segunda forma de utilizar la función ***plot()*** es con dos vectores como argumentos. En este caso los elementos del segundo vector se representan en ordenadas frente a los valores del primero, que se representan en abscisas. Véase por ejemplo cómo se puede dibujar un cuadrilátero de esta forma (obsérvese que para dibujar un polígono cerrado el último punto debe coincidir con el primero):

```
>> x=[1 6 5 2 1]; y=[1 0 4 3 1];
>> plot(x,y)
```

La función ***plot()*** permite también dibujar múltiples curvas introduciendo varias parejas de vectores como argumentos. En este caso, cada uno de los segundos vectores se dibujan en ordenadas como función de los valores del primer vector de la pareja, que se representan en abscisas. Si el usuario no decide otra cosa, para las sucesivas líneas se utilizan colores que son permutaciones cíclicas del ***azul***, ***verde***, ***rojo***, ***cyan***, ***magenta***, ***amarillo*** y ***negro***. Obsérvese bien cómo se dibujan el seno y el coseno en el siguiente ejemplo:

```
>> x=0:pi/25:6*pi;
>> y=sin(x); z=cos(x);
>> plot(x,y,x,z)
```

Ahora se va a ver lo que pasa con los **vectores complejos**. Si se pasan a ***plot()*** varios vectores complejos como argumentos, MATLAB simplemente representa las partes reales y desprecia las partes imaginarias. Sin embargo, un único argumento complejo hace que se represente la parte real en abscisas, frente a la parte imaginaria en ordenadas. Véase el siguiente ejemplo. Para generar un vector complejo se utilizará el resultado del cálculo de valores propios de una matriz formada aleatoriamente:

```
>> plot(eig(rand(20,20)),'+')
```

donde se ha hecho uso de elementos que se verán en la siguiente sección, respecto a dibujar con distintos tipos de "markers" (en este caso con signos +), en vez de con línea continua, que es la opción por defecto. En el comando anterior, el segundo argumento es un carácter que indica el tipo de marker elegido. El comando anterior es equivalente a:

```
>> z=eig(rand(20,20));
>> plot(real(z),imag(z),'+')
```

Como ya se ha dicho, si se incluye más de un vector complejo como argumento, se ignoran las partes imaginarias. Si se quiere dibujar varios vectores complejos, hay que separar explícitamente las partes reales e imaginarias de cada vector, como se acaba de hacer en el último ejemplo.

El comando ***plot*** puede utilizarse también con matrices como argumentos. Véanse algunos ejemplos sencillos:

plot(A)	dibuja una línea por cada columna de **A** en ordenadas, frente al índice de los elementos en abscisas
plot(x,A)	dibuja las columnas (o filas) de **A** en ordenadas frente al vector **x** en abscisas. Las dimensiones de **A** y **x** deben ser coherentes: si la matriz **A** es cuadrada se dibujan las columnas, pero si no lo es y la dimensión de las filas coincide con la de **x**, se dibujan las filas
plot(A,x)	análogo al anterior, pero dibujando las columnas (o filas) de **A** en abscisas, frente al valor de **x** en ordenadas
plot(A,B)	dibuja las columnas de **B** en ordenadas frente a las columnas de **A** en abscisas, dos a dos. Las dimensiones deben coincidir
plot(A,B,C,D)	análogo al anterior para cada par de matrices. Las dimensiones de cada par deben coincidir, aunque pueden ser diferentes de las dimensiones de los demás pares

Se puede obtener una excelente y breve descripción de la función ***plot()*** con el comando ***help plot*** o ***helpwin plot***. La descripción que se acaba de presentar se completará en la siguiente sección, en donde se verá cómo elegir los colores y los tipos de línea.

8.1.2. ESTILOS DE LÍNEA Y MARCADORES EN LA FUNCIÓN *PLOT*

En la sección anterior se ha visto cómo la tarea fundamental de la función ***plot()*** era dibujar los valores de un vector en ordenadas, frente a los valores de otro vector en abscisas. En el caso general esto exige que se pasen como argumentos un par de vectores. En realidad, el conjunto básico de argumentos de esta función es una *tripleta* formada por dos vectores y una cadena de 1, 2 ó 3 caracteres que indica el color y el tipo de línea o de marker. En la tabla siguiente se pueden observar las distintas posibilidades.

Símbolo	**Color**	**Símbolo**	**Marcadores (markers)**
y	yellow	.	puntos
m	magenta	o	círculos
c	cyan	x	marcas en x
r	red	+	marcas en +
g	green	*	marcas en *
b	blue	s	marcas cuadradas (square)
w	white	d	marcas en diamante (diamond)
k	black	^	triángulo apuntando arriba
		v	triángulo apuntando abajo
Símbolo	**Estilo de línea**	>	triángulo apuntando a la dcha
-	líneas continuas	<	triángulo apuntando a la izda
:	líneas a puntos	p	estrella de 5 puntas
-.	líneas a barra-punto	h	estrella se seis puntas
--	líneas a trazos		

Tabla 1. Colores, markers y estilos de línea.

Cuando hay que dibujar varias líneas, por defecto se van cogiendo sucesivamente los colores de la tabla comenzando por el azul, hacia arriba, y cuando se terminan se vuelve a empezar otra vez por el azul. Si el fondo es blanco, este color no se utiliza para las líneas.

También es posible añadir en la función ***plot*** algunos especificadores de línea que controlan el espesor de la línea, el tamaño de los marcadores, etc. Considérese el siguiente ejemplo:

```
plot(x,y,'-.rs', 'LineWidth',4, 'MarkerEdgeColor','k', 'MarkerFaceColor', 'g',...
                 'MarkerSize',10)
```

8.1.3. AÑADIR LÍNEAS A UN GRÁFICO YA EXISTENTE

Existe la posibilidad de añadir líneas a un gráfico ya existente, sin destruirlo o sin abrir una nueva ventana. Se utilizan para ello los comandos ***hold on*** y ***hold off***. El primero de ellos hace que los gráficos sucesivos respeten los que ya se han dibujado en la figura (es posible que haya que modificar la escala de los ejes); el comando ***hold off*** deshace el efecto de ***hold on***. El siguiente ejemplo muestra cómo se añaden las gráficas de **x2** y **x3** a la gráfica de **x** previamente creada (cada una con un tipo de línea diferente):

```
>> plot(x)
>> hold on
```

```
>> plot(x2,'--')
>> plot(x3,'-.')
>> hold off
```

8.1.4. COMANDO *SUBPLOT*

Una ventana gráfica se puede dividir en **m** particiones horizontales y **n** verticales, con objeto de representar múltiples gráficos en ella. Cada una de estas subventanas tiene sus propios ejes, aunque otras propiedades son comunes a toda la figura. La forma general de este comando es:

```
>> subplot(m,n,i)
```

donde **m** y **n** son el número de subdivisiones en filas y columnas, e **i** es la subdivisión que se convierte en activa. Las subdivisiones se numeran consecutivamente empezando por las de la primera fila, siguiendo por las de la segunda, etc. Por ejemplo, la siguiente secuencia de comandos genera cuatro gráficos en la misma ventana:

```
>> y=sin(x); z=cos(x); w=exp(-x*.1).*y; v=y.*z;
>> subplot(2,2,1), plot(x,y)
>> subplot(2,2,2), plot(x,z)
>> subplot(2,2,3), plot(x,w)
>> subplot(2,2,4), plot(x,v)
```

Se puede practicar con este ejemplo añadiendo títulos a cada *subplot*, así como rótulos para los ejes. Se puede intentar también cambiar los tipos de línea. Para volver a la opción por defecto basta teclear el comando:

```
>> subplot(1,1,1)
```

8.1.5. CONTROL DE LOS EJES: FUNCIÓN *AXIS()*

También en este punto MATLAB tiene sus opciones por defecto, que en algunas ocasiones puede interesar cambiar. El comando básico es el comando ***axis***. Por defecto, MATLAB ajusta la escala de cada uno de los ejes de modo que varíe entre el mínimo y el máximo valor de los vectores a representar. Este es el llamado modo "auto", o modo automático. Para definir de modo explícito los valores máximo y mínimo según cada eje, se utiliza el comando:

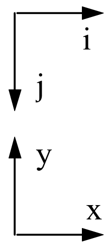
```
axis([xmin, xmax, ymin, ymax])
```

mientras que :

```
axis('auto')
```

devuelve el escalado de los ejes al valor por defecto o automático. Otros posibles usos de este comando son los siguientes:

v=axis	devuelve un vector **v** con los valores [xmin, xmax, ymin, ymax]
axis('ij')	utiliza *ejes de pantalla*, con el origen en la esquina superior izda. y el eje **j** en dirección vertical descendente
axis('xy')	utiliza *ejes cartesianos* normales, con el origen en la esquina inferior izda. y el eje **y** vertical ascendente
axis('auto x')	utiliza el escalado automático sólo en dirección *x*
axis('auto xz')	utiliza el escalado automático sólo en direcciones *x*, *z*
axis(axis)	mantiene los ejes en sus actuales valores, de cara a posibles nuevas gráficas añadidas con ***hold on***
axis('tight')	establece los mismos límites para los ejes que para los datos
axis('equal')	el escalado es igual en ambos ejes
axis('square')	la ventana será cuadrada



axis('image')	la ventana tendrá las proporciones de la imagen que se desea representar en ella (por ejemplo la de una imagen bitmap que se desee importar) y el escalado de los ejes será coherente con dicha imagen
axis('normal')	elimina las restricciones introducidas por 'equal' y 'square'
axis('off')	elimina las etiquetas, los números y los ejes
axis('on')	restituye las etiquetas, los números y los ejes
XLim, YLim	permiten modificar selectivamente los valores máximo y mínimo de los ejes en las direcciones *x* e *y*.

Es posible también tener un control preciso sobre las marcas y los rótulos que aparecen en los ejes, como por ejemplo en la forma siguiente:

```
x = -pi:.1:pi; y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

Obsérvese cómo las propiedades se establecen sobre los ejes actuales, a los que se accede con la función ***gca*** (*get current axis*).

8.1.6. FUNCIÓN *LINE()*

La función ***line()*** permite dibujar una o más líneas que unen los puntos cuyas coordenadas se pasan como argumentos. Permite además especificar el color, grosor, tipo de trazo, marcador, etc. Es una función de más bajo nivel que la función ***plot()***, pero ofrece una mayor flexibilidad. En su versión más básica, para dibujar un segmento de color verde entre dos puntos, esta función se llamaría de la siguiente manera:

```
>> line([xini, xend]', [yini, yend]', 'color', 'g')
```

Se puede también dibujar dos líneas a la vez utilizando la forma:

```
>> line([xini1 xini2; xend1 xend2], ([yini1 yini2; yend1 yend2]);
```

Finalmente, si cada columna de la matriz **X** contiene la coordenada *x* inicial y final de un punto, y lo mismo las columnas de la matriz **Y** con las coordenadas *y*, la siguiente sentencia dibuja tantas líneas como columnas tengan las matrices **X** e **Y**:

```
>> line([X], [Y]);
```

Se pueden controlar las características de la línea por medio de pares parámetro/valor, como por ejemplo:

```
>> line(x,y,'Color','r','LineWidth',4,'MarkerSize',12,'LineStyle','—','Marker','*')
```

8.1.7. FUNCIÓN *FINDOBJ()*

Si al dibujar una línea se recupera el valor de retorno de la función ***line*** y se almacena en una variable, más tarde es posible realizar un borrado selectivo de esa línea, como se ha comentado en el apartado 8.1. Sin embargo, aunque no se haya tomado esta precacución, también es posible recuperar la referencia (*handle*) a un determinado elemento gráfico de una figura por medio de la función ***findobj*** (*find object*), a la que se pasan ciertas características del elemento gráfico que permiten su localización.

Algunos posibles usos de esta función son los siguientes:

```
>> h=findobj
```

recupera la referencia del objeto base de la jerarquía gráfica y de todos sus descendientes;

```
>> findobj('color','k')
```

devuelve las referencias de todos los objetos de color negro;

```
>> set(findobj(gca,'Type','line','Color','b'), 'Color','r',     'LineWidth',3)
```

encuentra los objetos de tipo línea y de color azul, y cambia el color a rojo a la vez que establece una anchura de línea de tres pixels.

8.2. Control de ventanas gráficas: Función *figure*

Si se llama a la función ***figure*** sin argumentos, se crea una nueva ventana gráfica con el número consecutivo que le corresponda. El valor de retorno es dicho número.

Por otra parte, el comando ***figure(n)*** hace que la ventana **n** pase a ser la ventana o figura activa. Si dicha ventana no existe, se crea una nueva ventana con el número consecutivo que le corresponda (que se puede obtener como valor de retorno del comando). La función ***close*** cierra la figura activa, mientras que ***close(n)*** cierra la ventana o figura número **n**.

El comando ***clf*** elimina el contenido de la figura activa, es decir, la deja abierta pero vacía. La función ***gcf*** devuelve el número de la figura activa en ese momento.

Para practicar un poco con todo lo que se acaba de explicar, ejecútense las siguientes instrucciones de MATLAB, observando con cuidado los efectos de cada una de ellas en la ventana activa. El comando ***figure(gcf)*** (*get current figure*) permite hacer visible la ventana de gráficos desde la ventana de comandos.

```
>> x=[-4*pi:pi/20:4*pi];
>> plot(x,sin(x),'r',x,cos(x),'g')
>> title('Función seno(x) -en rojo- y función coseno(x) -en verde-')
>> xlabel('ángulo en radianes'), figure(gcf)
>> ylabel('valor de la función trigonométrica'), figure(gcf)
>> axis([-12,12,-1.5,1.5]), figure(gcf)
>> axis('equal'), figure(gcf)
>> axis('normal'), figure(gcf)
>> axis('square'), figure(gcf)
>> axis('off'), figure(gcf)
>> axis('on'), figure(gcf)
>> axis('on'), grid, figure(gcf)
```

La función ***figure*** también admite que se fijen algunas de sus propiedades, como por ejemplo la posición y el tamaño con que aparecerá en la pantalla. Por ejemplo, el comando:

```
>> figure('position',[left,botton, width,height])
```

abre una ventana cuya esquina inferior izquierda está en el punto (**left,botton**) respecto a la esquina inferior izquierda de la pantalla (en pixels), que tiene una anchura de **width** pixels y una altura de **height** pixels.

Otra característica muy importante de una ventana gráfica es la de representar animaciones utilizando la técnica del ***doble buffer***. De modo sencillo, esta técnica se puede explicar diciendo que es como si el ordenador tuviera dos paneles de dibujo: mientras uno está a la vista, se está dibujando en el otro, y cuando el dibujo está terminado este segundo panel se hace visible. El resultado del doble buffer es que las animaciones y el movimiento se ven de modo perfecto, sin el ***parpadeo*** (*flicker*) tan característico cuando no se utiliza esta técnica.

Para dibujar con doble buffer en la ventana activa basta ejecutar los comandos siguientes (sin demasiadas explicaciones, que se pueden buscar en el ***Help*** de MATLAB):

```
>> set(gcf,'DoubleBuffer','on', 'Renderer','painters')
```

8.3. Otras funciones gráficas 2-D

Existen otras funciones gráficas bidimensionales orientadas a generar otro tipo de gráficos distintos de los que produce la función ***plot()*** y sus análogas. Algunas de estas funciones son las siguientes (para más información sobre cada una de ellas en particular, utilizar ***help nombre_función***):

bar()	crea diagramas de barras
barh()	diagramas de barras horizontales
bar3()	diagramas de barras con aspecto 3-D
bar3h()	diagramas de barras horizontales con aspecto 3-D
pie()	gráficos con forma de "tarta"
pie3()	gráficos con forma de "tarta" y aspecto 3-D
area()	similar ***plot()***, pero rellenando en ordenadas de 0 a y
stairs()	función análoga a ***bar()*** sin líneas internas
errorbar()	representa sobre una gráfica –mediante barras– valores de errores
compass()	dibuja los elementos de un vector complejo como un conjunto de vectores partiendo de un origen común
feather()	dibuja los elementos de un vector complejo como un conjunto de vectores partiendo de orígenes uniformemente espaciados sobre el eje de abscisas
hist()	dibuja histogramas de un vector
rose()	histograma de ángulos (en radianes)
quiver()	dibujo de campos vectoriales como conjunto de vectores

Por ejemplo, genérese un vector de valores aleatorios entre 0 y 10, y ejecútense los comandos:

```
>> x=[rand(1,100)*10];
>> plot(x)
>> bar(x)
>> stairs(x)
>> hist(x)
>> hist(x,20)
>> alfa=(rand(1,20)-0.5)*2*pi;
>> rose(alfa)
```

8.3.1. FUNCIÓN *FPLOT*

La función ***plot*** vista anteriormente dibuja vectores. Si se quiere dibujar una función, antes de ser pasada a ***plot*** debe ser convertida en un vector de valores. Esto tiene algunos inconvenientes, por ejemplo, el que "a priori" es difícil predecir en que zonas la función varía más rápidamente y habría por ello que reducir el espaciado entre los valores en el eje de abscisas.

La función ***fplot*** admite como argumento un *nombre de función* o un *nombre de fichero* ****.m*** en el cual esté definida una función de usuario. La función puede ser escalar (un único resultado por cada valor de **x**) o vectorial. La forma general de esta función es la siguiente:

```
fplot('funcion', limites, 'cadena', tol)
```

donde:

'funcion'	representa el nombre de la función o del fichero ****.m*** entre apóstrofos (pasado como cadena de caracteres),
limites	es un vector de 2 ó 4 elementos, cuyos valores son [**xmin,xmax**] o [**xmin,xmax,ymin,ymax**],
'cadena'	tiene el mismo significado que en ***plot*** y permite controlar el color, los markers y el tipo de línea.

tol es la tolerancia de error relativo. El valor por defecto es 2e-03. El máximo número de valores en **x** es (1/tol)+1

Esta función puede utilizarse también en la forma:

```
[x,y]=fplot('funcion', limites, 'cadena', tol)
```

y en este caso se devuelven los vectores **x** e **y**, pero no se dibuja nada. El gráfico puede obtenerse con un comando posterior por medio de la función ***plot***. Véase un ejemplo de utilización de esta función. Se comienza creando un fichero llamado ***mifunc.m*** en el directorio ***G:\matlab*** que contenga las líneas siguientes:

```
function y = mifunc(x)
y(:,1)=200*sin(x)./x;
y(:,2)=x.^2;
```

y a continuación se ejecuta el comando:

```
>> fplot('mifunc(x)', [-20 20], 'g')
```

Obsérvese que la función ***mifunc*** devuelve una matriz con dos columnas, que constituyen las dos gráficas dibujadas. En este caso se ha utilizado para ellas el color verde.

8.3.2. FUNCIÓN *FILL* PARA POLÍGONOS

Ésta es una función especial para dibujar polígonos planos, rellenándolos de un determinado color. La forma general es la siguiente:

```
>> fill(x,y,c)
```

que dibuja un polígono definido por los vectores **x** e **y**, rellenándolo con el color especificado por **c**. Si es necesario, el polígono se cierra uniendo el último vértice con el primero. Respecto al color:

– Si **c** es un carácter de color ('r','g','b','c','m','y','w','k'), o un vector de valores [r g b], el polígono se rellena de modo uniforme con el color especificado.

– Si **c** es un vector de la misma dimensión que **x** e **y**, sus elementos se trasforman de acuerdo con un mapa de colores determinado, y el llenado del polígono –no uniforme en este caso– se obtiene interpolando entre los colores de los vértices. Sobre este tema de los colores, se volverá más adelante con un cierto detenimiento.

Este comando puede utilizarse también con matrices:

```
>> fill(A,B,C)
```

donde **A** y **B** son matrices del mismo tamaño. En este caso se dibuja un polígono por cada par de columnas de dichas matrices. **C** puede ser un vector fila de colores uniformes para cada polígono, o una matriz del mismo tamaño que las anteriores para obtener colores de relleno por interpolación. Si una de las dos, o **A** o **B**, son un vector en vez de una matriz, se supone que ese vector se repite tantas veces como sea necesario para dibujar tantos polígonos como columnas tiene la matriz. Considérese un ejemplo sencillo de esta función:

```
>> x=[1 5 4 2]; y=[1 0 4 3];
>> fill(x,y,'r')
>> colormap(gray), fill(x,y,[1 0.5 0.8 0.7])
```

8.3.3. DIBUJO SIMPLIFICADO DE FUNCIONES: FUNCIONES *EZPLOT()* Y *EZPOLAR()*

La función ***ezplot*** es una función de dibujo simplificada, útil cuando se quiere obtener de forma muy rápida la gráfica de una función. En su forma más simple, se puede llamar en la forma:

```
>> ezplot(f);
```

donde **f** es el nombre o mejor el handle de una función. También puede ser una ***función inline***. Por defecto la función se dibuja en el intervalo [$-2\pi \leq x \leq 2\pi$]. Si se desea dibijar f en un intervalo diferente, se puede escribir:

```
>> ezplot(f,[a,b]);
```

La función **f** puede ser una ***función implícita de dos variables*** $f(x,y)=0$. El intervalo por defecto para cada variable es [$-2\pi \leq x \leq 2\pi$]. También se puede definir un intervalo común o específico para cada variable.

```
>> ezplot(f);         % dibuja f(x,y)=0 en -2*pi<x<2*pi y -2*pi<y<2*pi
>> ezplot(f, [a,b]);  % dibuja f(x,y)=0 en a<x<b y a<y<b
>> ezplot(f, [xmin,xmax,ymin,ymax]);
```

La función ***ezplot*** puede dibujar también ***funciones paramétricas*** $x(t)$, $y(t)$, como por ejemplo:

```
>> ezplot('sin(t)','cos(t)');           % dibuja para 0<t<2*pi
>> ezplot('sin(t)','cos(t)', [t1,t2]);  % dibuja para t1<t<t2
>> f = inline('cos(x)+2*sin(2*x)'); ezplot(f);
```

La función ***ezpolar*** es similar a ***ezplot*** y se utiliza para dibujar en ***coordenadas polares***.

8.4. Entrada de puntos con el ratón

Se realiza mediante la función ***ginput***, que permite introducir las coordenadas del punto sobre el que está el cursor, al clicar (o al pulsar una tecla). Algunas formas de utilizar esta función son las siguientes:

[x,y] = ginput	lee un número indefinido de puntos –cada vez que se clica o se pulsa una tecla cualquiera– hasta que se termina pulsando la tecla ***intro***
[x,y] = ginput(n)	lee las coordenadas de **n** puntos
[x,y,bot] = ginput	igual que el anterior, pero devuelve también un vector de enteros **bot** con el código ASCII de la tecla pulsada o el número del botón del ratón (1, 2, ...) con el que se ha clicado

Como ejemplo de utilización de este comando, ejecútense las instrucciones siguientes en la ventana de comandos de MATLAB para introducir un cuadrilátero arbitrario y dibujarlo de dos formas:

```
>> clf, [x,y]=ginput(4);
>> figure(gcf), plot(x,y,'w'), pause(5), fill(x,y,'r')
```

donde se ha introducido el comando ***pause(5)*** que espera 5 segundos antes de continuar la ejecución. Este comando admite como argumento un tiempo con precisión de centésimas de segundo.

8.5. Preparación de películas o "movies"

Para preparar pequeñas películas o movies se pueden utilizar las funciones ***movie***, ***moviein*** y ***getframe***. Una película se compone de varias imágenes, denominadas *frames*. La función ***getframe*** devuelve un vector columna con la información necesaria para reproducir la imagen que se acaba de representar en la figura o ventana gráfica activa, por ejemplo con la función ***plot***. El tamaño de este vector columna depende del tamaño de la ventana, pero no de la complejidad del dibujo. La función ***moviein(n)*** reserva memoria para almacenar **n** frames. La siguiente lista de comandos crearía una película de 17 imágenes o frames, que se almacenarán como las columnas de la matriz **M**:

```
M = moviein(17);
x=[-2*pi:0.1:2*pi]';
for j=1:17
    y=sin(x+j*pi/8);
    plot(x,y);
    M(:,j) = getframe;
end
```

Una vez creada la película se puede representar el número de veces que se desee con el comando ***movie***. Por ejemplo, para representar 10 veces la película anterior, a 15 imágenes por segundo, habría que ejecutar el comando siguiente (los dos últimos parámetros son opcionales):

```
movie(M,10,15)
```

Los comandos ***moviein***, ***getframe*** y ***movie*** tienen posibilidades adicionales para las que puede consultarse el ***Help*** correspondiente. Hay que señalar que en MATLAB no es lo mismo un ***movie*** que una ***animación***. Una *animación* es simplemente una ventana gráfica que va cambiando como consecuencia de los comandos que se van ejecutando. Un *movie* es una animación grabada o almacenada en memoria previamente.

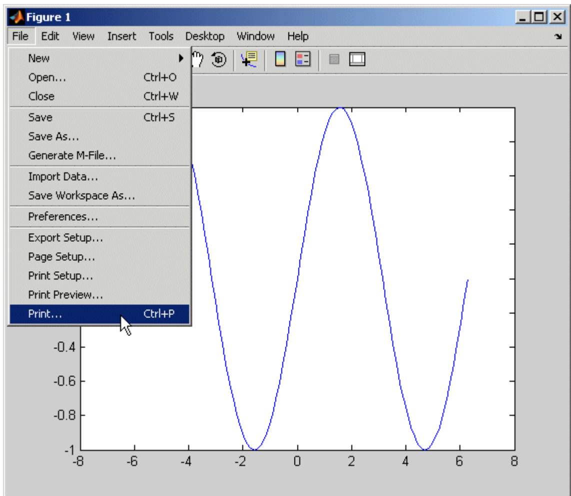


Figura 39. Comandos para imprimir figuras.

8.6. Impresión de las figuras en impresora láser

Es muy fácil enviar a la impresora o a un fichero una figura producida con MATLAB. La **¡Error! No se encuentra el origen de la referencia.** muestra las opciones que ofrece el menú ***File*** relacionadas con la impresión de figuras: es posible establecer los parámetros de la página (***Page Setup***), de la impresora (***Print Setup***), obtener una visión preliminar (***Print Preview***) e imprimir (***Print***). Todos estos comandos se utilizan en la forma habitual de las aplicaciones de ***Windows***.

La opción ***Page Setup*** abre el cuadro de diálogo de la Figura 40, que permite situar el dibujo sobre la página, establecer los márgenes, la orientación del papel, etc.

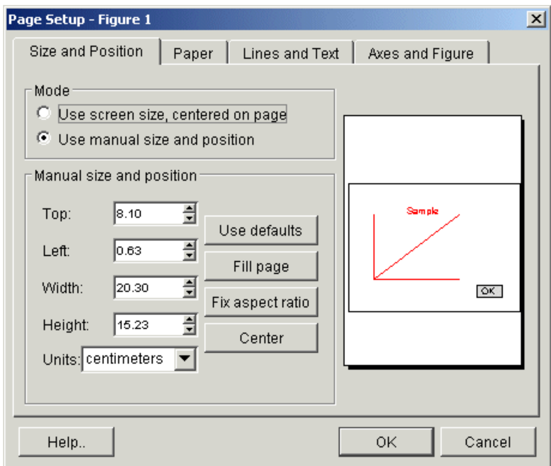


Figura 40. Preparar la impresión con *Page Setup*.

La impresión de una figura puede hacerse también desde la línea de comandos. La forma general del comando de impresión es la siguiente (si se omite el nombre del fichero, la figura se envía a la impresora):

```
>> print -device -options filename
```

Mediante el ***Help*** se puede obtener más información sobre el comando ***print***.

Es posible también exportar a un fichero una figura de MATLAB, por ejemplo para incluirla luego en un documento de ***Word*** o en una presentación de ***Powerpoint***. Para ello se utiliza el comando ***File/Save as*** de la ventana en la que aparece la figura. El cuadro de diálogo que se abre ofrece distintos formatos gráficos para guardar la imagen. Cabe destacar la ausencia del formato ****.gif***, muy utilizado en Internet; sí está presente sin embargo el

formato ***.png**, que se considera el sucesor natural del ***.gif**. En todo caso la figura puede exportarse con cualquier formato estándar y luego utilizar por ejemplo ***Paint Shop Pro*** para transformarla.

8.7. Las ventanas gráficas de MATLAB

Anteriormente han aparecido en varias ocasiones las ventanas gráficas de MATLAB. Quizás sea el momento de hacer una breve recapitulación sobre sus posibilidades, que se han ido mejorando en las sucesivas versiones. La Figura 41 muestra los menús y las barras de herramientas de las ventanas gráficas de MATLAB. Por defecto sólo aparece la barra de herramientas de la línea superior. Para hacer aparecer también la segunda barra se ejecuta ***Camera Toolbar***, en el menú ***View***.

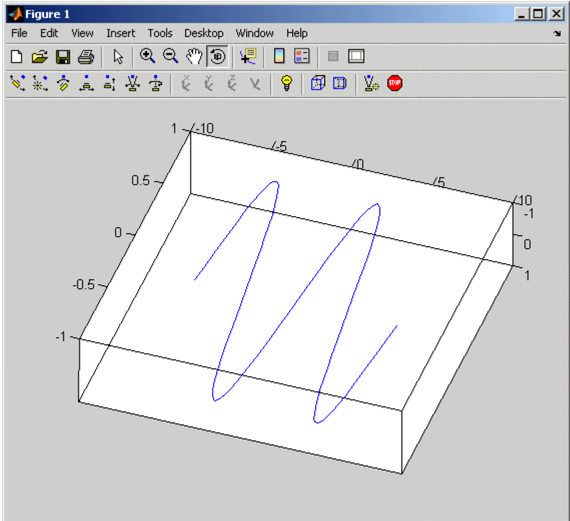


Figura 41. Menús y barras de las ventanas gráficas.

En el menú ***Edit***, además de los comandos referentes a la copia de figuras, aparecen los comandos ***Figure Properties***, ***Axes Properties***, ***Current Object Properties*** y ***Colormap,*** que abren paso a los correspondientes editores de propiedades. Los tres primeros se muestran en las figuras siguientes (con la parte de la imagen con tamaño reducido).

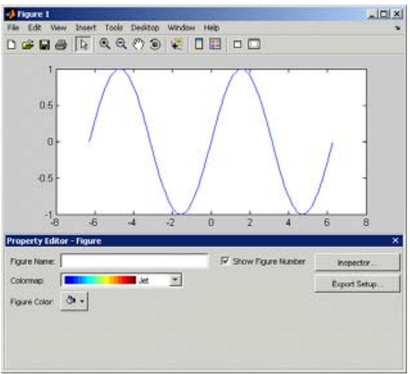


Figura 42. Editor de propiedades de Figure.

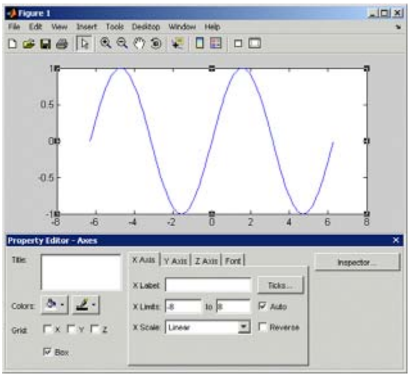


Figura 43. Editor de propiedades de Axes.

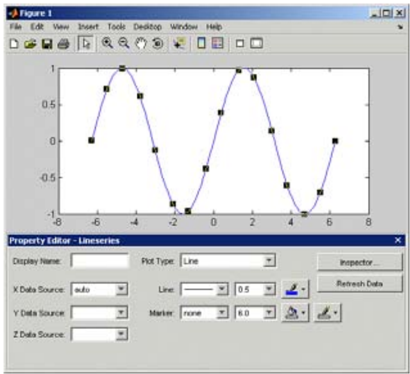


Figura 44. Editor de propiedades de objeto (una línea).

También es posible cambiar interactivamente el ***mapa de colores*** utilizado en una figura (ver apartado 9.2.1). Con el comando ***Edit/Colormap*** se abre el cuadro de diálogo mostrado en la Figura 45.

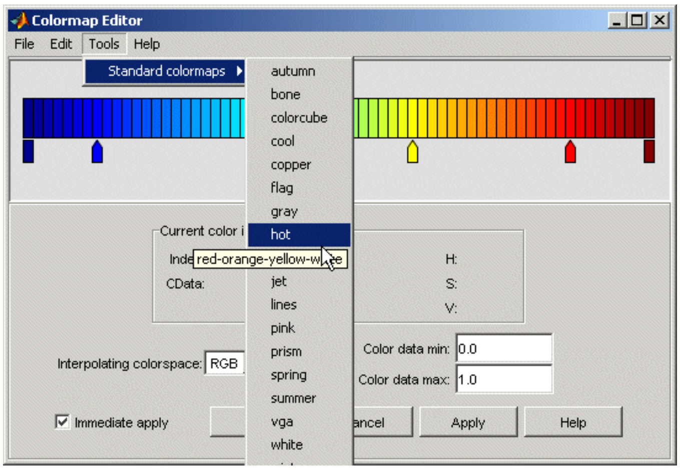


Figura 45. Editor de mapa de colores.

El menú ***Edit*** de las ventanas gráficas ofrece también las opciones estándar de ***Windows***, permitiendo copiar, cortar y pegar los elementos seleccionados de la figura si está activada la opción ***Plot Edit*** ( ).

El menú ***View*** permite hacer visibles u ocultar las barras de herramientas ***Window Toolbar*** y ***Camera Toolbar***. Como se ha dicho, por de-

fecto sólo aparece la primera de ellas. Estas barras de herramientas disponen de numerosas opciones para trabajar con ventanas que contengan gráficos 2-D y 3-D. La mejor forma de aprender es probar y acudir al ***Help*** cuando hay algo que no se entiende.

En menú ***Insert*** permite añadir elementos a la figura activa, por ejemplo rótulos, etiquetas, líneas, texto, etc. Por su parte, el menú ***Tools*** permite realizar desde menú algunas de las operaciones también disponibles en las barras de herramientas. Finalmente, el menú ***Help*** permite acceder a la ayuda concreta que hace referencia a las ventanas gráficas.

9. GRÁFICOS TRIDIMENSIONALES

Quizás sea ésta una de las características de MATLAB que más admiración despierta entre los usuarios no técnicos (cualquier alumno de ingeniería sabe que hay ciertas operaciones algebraicas –como la descomposición de valores singulares, sin ir más lejos– que tienen dificultades muy superiores, aunque "luzcan" menos).

9.1. Tipos de funciones gráficas tridimensionales

MATLAB tiene posibilidades de realizar varios tipos de gráficos 3D. Para darse una idea de ello, lo mejor es verlo en la pantalla cuanto antes, aunque haya que dejar las explicaciones detalladas para un poco más adelante.

La primera forma de gráfico 3D es la función ***plot3***, que es el análogo tridimensional de la función ***plot***. Esta función dibuja puntos cuyas coordenadas están contenidas en 3 vectores, bien uniéndolos mediante una línea continua (defecto), bien mediante ***markers***. Asegúrese de que no hay ninguna ventana gráfica abierta y ejecute el siguiente comando que dibuja una línea espiral en color rojo:

```
>> fi=[0:pi/20:6*pi]; plot3(cos(fi),sin(fi),fi,'r'), grid
```

Ahora se verá cómo se representa una función de dos variables. Para ello se va a definir una función de este tipo en un fichero llamado ***test3d.m***. La fórmula será la siguiente:

$$z = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2}$$

El fichero ***test3d.m*** debe contener las líneas siguientes:

```
function z=test3d(x,y)
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Ahora, ejecútese la siguiente lista de comandos (directamente, o mejor creando un fichero llamado ***test3dMain.m*** que los contenga):

```
>> x=[-3:0.4:3]; y=x;
>> close
>> subplot(2,2,1)
>> figure(gcf),fi=[0:pi/20:6*pi];
>> plot3(cos(fi),sin(fi),fi,'r')
>> grid
>> [X,Y]=meshgrid(x,y);
>> Z=test3d(X,Y);
>> subplot(2,2,2)
>> figure(gcf), mesh(Z)
>> subplot(2,2,3)
>> figure(gcf), surf(Z)
>> subplot(2,2,4)
>> figure(gcf), contour3(Z,16)
```

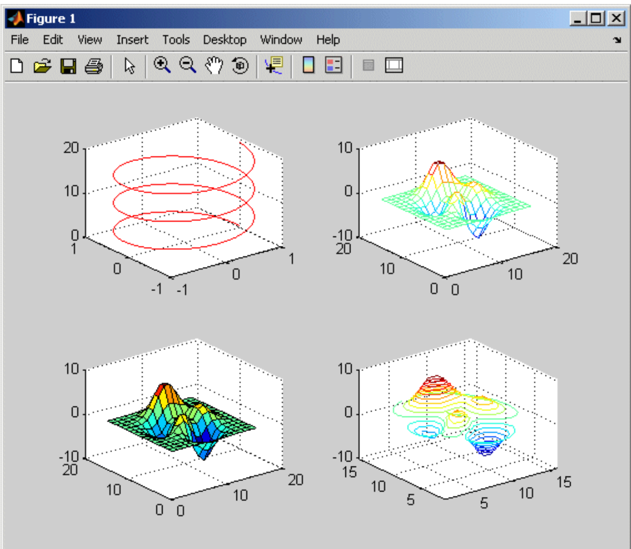


Figura 46. Gráficos 3D realizados con MATLAB.

En la figura resultante (Figura 46) aparece una buena muestra de algunas de las posibilidades gráficas tridimensionales de MATLAB. En las próximas secciones se realizará una explicación más detallada de qué se ha hecho y cómo se ha hecho.

9.1.1. DIBUJO SIMPLIFICADO DE FUNCIONES 3-D: FUNCIONES *EZPLOT3()*, *EZSURF()*, ETC.

Existen también algunas funciones simplificadas para el dibujo 3-D similares a la función ***ezplot*** vista en el Apartado 8.3.3, en la página 116.

Así la función ***ezplot3*** dibuja lineas paramétricas tridimensionales en la forma $x(t)$, $y(t)$ y $z(t)$. Por defecto se utiliza el intervalo $0 < t < 2$*pi. Considérense las siguientes posibilidades:

```
>> ezplot3(x,y,z);
>> ezplot3(x,y,z,[t1,t2]);
>> ezplot3(x,y,z,[t1,t2],'animate'); % dibuja la curva progresivamente
```

En las sentencias anteriores *x*, *y*, y *z* pueden ser funciones anónimas, handles a funciones, funciones inline o expresiones definidas como cadena de caracteres. Los ficheros *.m y las funciones inline deben escribirse de tal forma que admitan vectores de valores como argumentos (vectorizados).

Otra función de dibujo 3-D rápido es ***ezsurf***. Esta función utiliza la funcción ***surf*** para realizar un dibujo 3-D de una función $f(x,y)$. Por defecto se utilizan los intervalos –2*pi < *x*, *y* < 2*pi. La función *f* se puede definir por medio de una expresión en la que aparezcan *x* e *y* definida por medio de una cadena de caracteres, con una función convencional, o con funciones anónimas u online. A continuación se dan algunas posibles formas de exta función:

```
>> ezsurf(f);
>> ezsurf(f, [a,b]);
>> ezsurf(f, [xmin,xmax,ymin,ymax]);
```

La función ezsurf permite también dibujar ***superficies paramétricas*** 3-D, por ejmplo en las formas siguientes, con parámetros *s* y *t*:

```
>> ezsurf(x,y,z); % por defecto -2*pi < s,t < 2*pi
>> ezsurf(x,y,z, [a,b]);
>> ezsurf(x,y,z, [smin,smax,tmin,tmax]);
```

Con un último parámetro entero N se puede controlar la densidad del mallado con el que se dibuja. Por defecto N=60. Con el argumento 'circ' se dibuja en un dominio circular. A continuación se incluyen algunos ejemplos tomados de la ayuda de MATLAB:

```
>> ezsurf('s*cos(t)','s*sin(t)','t')
>> ezsurf('s*cos(t)','s*sin(t)','s')
>> ezsurf('exp(-s)*cos(t)','exp(-s)*sin(t)','t',[0,8,0,4*pi])
```

Otras funciones simplificadas para dibujo 3-D son ***ezcontour***, ***ezcontourf***, ***ezmesh***, ***ezsurfc*** y ***ezmeshc***. Para más información consultar el ***Help*** de MATLAB.

9.1.2. DIBUJO DE LÍNEAS: FUNCIÓN *PLOT3*

La función ***plot3*** es análoga a su homóloga bidimensional ***plot***. Su forma más sencilla es:

```
>> plot3(x,y,z)
```

que dibuja una línea que une los puntos ($x(1)$, $y(1)$, $z(1)$), ($x(2)$, $y(2)$, $z(2)$), etc. y la proyecta sobre un plano para poderla representar en la pantalla. Al igual que en el caso plano, se puede incluir una cadena de 1, 2 ó 3 caracteres para determinar el color, los markers, y el tipo de línea:

```
>> plot3(x,y,z,s)
```

También se pueden utilizar tres matrices **X**, **Y** y **Z** del mismo tamaño:

```
>> plot3(X,Y,Z)
```

en cuyo caso se dibujan tantas líneas como columnas tienen estas 3 matrices, cada una de las cuales está definida por las 3 columnas homólogas de dichas matrices.

A continuación se va a realizar un ejemplo sencillo consistente en dibujar un *cubo*. Para ello se creará un fichero llamado ***cubo.m*** que contenga las aristas correspondientes, definidas mediante los vértices del cubo como una línea poligonal continua (obsérvese que algunas aristas se dibujan dos veces). El fichero ***cubo.m*** define una matriz **A** cuyas columnas son las coordenadas de los vértices, y cuyas filas son las coordenadas **x**, **y** y **z** de los mismos. A continuación incluye la llamada a la función ***plot3***:

```
% fichero cubo.m
close all
A=[0 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0
   0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1 1
   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0];
plot3(A(1,:)',A(2,:)',A(3,:)')
```

9.1.3. Dibujo de mallados: Funciones *meshgrid*, *mesh* y *surf*

Ahora se verá con detalle cómo se puede dibujar una ***función de dos variables*** ($z=f(x,y)$) sobre un dominio rectangular. Se verá que también se pueden dibujar los elementos de una matriz como función de los dos índices.

Sean **x** e **y** dos vectores que contienen las coordenadas en una y otra dirección de la retícula (*grid*) sobre la que se va a dibujar la función. Después hay que crear dos matrices **X** (cuyas filas son copias de **x**) e **Y** (cuyas columnas son copias de **y**). Estas matrices se crean con la función ***meshgrid***. Estas matrices representan respectivamente las coordenadas *x* e *y* de todos los puntos de la retícula. La matriz de valores **Z** se calcula a partir de las matrices de coordenadas **X** e **Y**. Finalmente hay que dibujar esta matriz **Z** con la función ***mesh***, cuyos elementos son función elemento a elemento de los elementos de **X** e **Y**. Véase como ejemplo el dibujo de la función ***sen(r)/r*** (siendo $r=sqrt(x^2+y^2)$; para evitar dividir por 0 se suma al denominador el número pequeño ***eps***). Para distinguirla de la función ***test3d*** anterior se utilizará **u** y **v** en lugar de **x** e **y**. Créese un fichero llamado ***sombrero.m*** que contenga las siguientes líneas:

```
close all
u=-8:0.5:8; v=u;
[U,V]=meshgrid(u,v);
R=sqrt(U.^2+V.^2)+eps;
W=sin(R)./R;
mesh(W)
```

Ejecutando este fichero se obtiene el gráfico mostrado en la Figura 47.

Se habrá podido comprobar que la función ***mesh*** dibuja *en perspectiva* una función en base a una retícula de líneas de colores, rodeando cuadriláteros del color de fondo, con eliminación de líneas ocultas. Más adelante se verá cómo controlar estos colores que aparecen. Baste decir por ahora que el color depende del valor **z** de la función. Ejecútese ahora el comando:

```
>> surf(W)
```

y obsérvese la diferencia en la Figura 48. En vez de líneas aparece ahora una superficie *faceteada*, también con eliminación de líneas ocultas. El color de las facetas depende también del valor de la función.

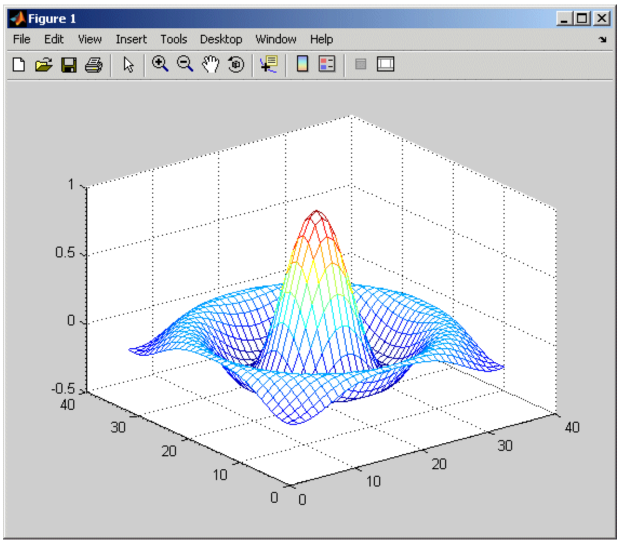


Figura 47. Figura 3D de la función "sombrero".

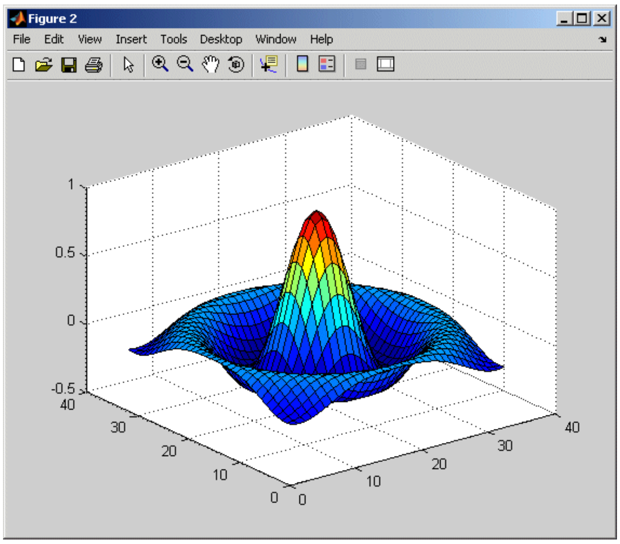


Figura 48. Función "sombrero" con facetas.

Como un segundo ejemplo, se va a volver a dibujar la función ***picos*** (la correspondiente al fichero ***test3d.m*** visto previamente). Créese ahora el fichero ***picos.m*** con las siguientes sentencias:

```
x=[-3:0.2:3];
y=x;
[X,Y]=meshgrid(x,y);
Z=test3d(X,Y);
figure(gcf), mesh(Z), pause(5), surf(Z)
```

Es necesario poner la instrucción ***pause*** –que espera 5 segundos– para que se puedan ver las dos formas de representar la función **Z** (si no, sólo se vería la segunda). Una vez creado este fichero, tecléese ***picos*** en la línea de comandos y obsérvese el resultado. Más adelante se verá también cómo controlar el punto de vista en estos gráficos en perspectiva.

9.1.4. DIBUJO DE LÍNEAS DE CONTORNO: FUNCIONES *CONTOUR* Y *CONTOUR3*

Una forma distinta de representar funciones tridimensionales es por medio de ***isolíneas*** o ***curvas de nivel***. A continuación se verá cómo se puede utilizar estas representaciones con las matrices de datos **Z** y **W** que se han calculado previamente:

```
>> contour(Z,20)
>> contour3(Z,20)
>> contour(W,20)
>> contour3(W,20)
```

donde "20" representa el número de líneas de nivel. Si no se pone se utiliza un número por defecto. Otras posibles formas de estas funciones son las siguientes:

contour(Z, val)	siendo **val** un vector de valores para las isolíneas a dibujar
contour(u,v,W,20)	se utilizan **u** y **v** para dar valores a los ejes de coordenadas
contour(Z,20,'r--')	se puede especificar el tipo de línea como en la función ***plot***
contourf(Z, val)	análoga a ***contour()***, pero rellenando el espacio entre líneas

9.2. Utilización del color en gráficos 3-D

En los dibujos realizados hasta ahora, se ha visto que el resultado adoptaba determinados colores, pero todavía no se ha explicado de dónde han salido. Ahora se verá qué sistema utiliza MATLAB para determinar los colores.

9.2.1. MAPAS DE COLORES

Un ***mapa de colores*** se define como una matriz de tres columnas, cada una de las cuales contiene un valor entre 0 y 1, que representa la intensidad de uno de los colores fundamentales: R (red o rojo), G (green o verde) y B (blue o azul).

La longitud por defecto de los mapas de colores de MATLAB es 64, es decir, cada mapa de color contiene 64 colores. Esta longitud puede modificarse como luego se verá.

Algunos mapas de colores están predefinidos en MATLAB. Buscando ***colormap*** en ***Help*** se obtiene –entre otra información– la lista de los siguientes mapas de colores:

autumn	varies smoothly from red, through orange, to yellow.
bone	is a grayscale colormap with a higher value for the blue component.
colorcube	contains as many regularly spaced colors in RGB colorspace as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
cool	consists of colors that are shades of cyan and magenta.
copper	varies smoothly from black to bright copper.
flag	consists of the colors red, white, blue, and black.
gray	returns a linear grayscale colormap.
hot	varies smoothly from black, through shades of red, orange, and yellow, to white.
hsv	varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red.
jet	ranges from blue to red, and passes through the colors cyan, yellow, and orange. It is a variation of the hsv colormap.
lines	colormap of colors specified by the Axes ColorOrder property and a shade of gray.
pink	contains pastel shades of pink.
prism	repeats the six colors red, orange, yellow, green, blue, and violet.
spring	consists of colors that are shades of magenta and yellow.
summer	consists of colors that are shades of green and yellow.
white	is an all white monochrome colormap.
winter	consists of colors that are shades of blue and green.

El colormap por defecto es ***jet***. Para visualizar estos mapas de colores, cambiando al mismo tiempo su longitud, se pueden utilizar los siguientes comandos en la ***Command Window***:

```
>> colormap(hot(128))
>> pcolor([1:129;1:129]')
```

donde la función ***pcolor*** permite visualizar por medio de colores la magnitud de los elementos de una matriz (en realidad representa colores de “celdas”, para lo que necesita que la matriz tenga una fila y columna más de las necesarias; ésa es la razón de que en el ejemplo anterior a la función ***pcolor*** se le pasen 129 filas y 2 columnas).

Si se desea imprimir una figura en una impresora láser en blanco y negro, puede utilizarse el mapa de color ***gray***. En el siguiente apartado se explica con más detalle el dibujo en "pseudocolor" (*pcolor*, abreviadamente).

El comando ***colormap*** actúa sobre la figura activa, cambiando sus colores. Si no hay ninguna figura activa, sustituye al mapa de color anterior para las siguientes figuras que se vayan a dibujar.

9.2.2. IMÁGENES Y GRÁFICOS EN *PSEUDOCOLOR*. FUNCIÓN *CAXIS*

Cuando se desea dibujar una figura con un determinado mapa de colores se establece una correspondencia (o un *mapping*) entre los valores de la función y los colores del mapa de colores. Esto hace que los valores pequeños se dibujen con los colores *bajos* del mapa, mientras que los valores grandes se dibujan con los colores *altos*.

La función ***pcolor*** es -en cierta forma- equivalente a la función ***surf*** con el punto de vista situado perpendicularmente al dibujo. Un ejemplo interesante de uso de la función ***pcolor*** es el siguiente: se

genera una matriz **A** de tamaño 100×100 con valores aleatorios entre 0 y 1. La función ***pcolor(A)*** dibuja en color los elementos de la matriz **A**, mientras que la función ***pcolor(inv(A))*** dibuja los colores correspondientes a los elementos de la matriz inversa. Se puede observar que los colores de la matriz inversa son mucho más uniformes que los de la matriz original. Los comandos son los siguientes:

```
>> A=rand(100,100); colormap(hot); pcolor(A); pause(5), pcolor(inv(A));
```

donde el comando ***pause(5)*** simplemente introduce un pausa de 5 seg en la ejecución. Al ejecutar todos los comandos en la misma línea es necesario poner ***pause*** pues si no dibuja directamente la inversa sin pasar por la matriz inicial.

Si todavía se conservan las matrices **Z** y **W** que se han definido previamente, se pueden hacer algunas pruebas cambiando el mapa de colores.

La función ***caxis*** permite ajustar manualmente la escala de colores. Su forma general es:

```
caxis([cmin, cmax])
```

donde ***cmin*** y ***cmax*** son los valores numéricos a los que se desea ajustar el mínimo y el máximo valor de la escala de colores.

9.2.3. DIBUJO DE SUPERFICIES FACETEADAS

La función ***surf*** tiene diversas posibilidades referentes a la forma en que son representadas las facetas o polígonos coloreados. Las tres posibilidades son las siguientes:

shading flat	determina sombreado con color constante para cada polígono. Este sombreado se llama plano o *flat*.
shading interp	establece que el sombreado se calculará por interpolación de colores entre los vértices de cada faceta. Se llama también sombreado de Gouraud
shading faceted	consiste en sombreado constante con líneas negras superpuestas. Esta es la opción por defecto

Edite el fichero ***picos.m*** de forma que aparezcan menos facetas y más grandes. Se puede probar con ese fichero, eliminando la función ***mesh***, los distintos tipos de sombreado o *shading* que se acaban de citar. Para obtener el efecto deseado, basta poner la sentencia ***shading*** a continuación de la sentencia ***surf***.

9.2.4. OTRAS FORMAS DE LAS FUNCIONES *MESH* Y *SURF*

Por defecto, las funciones ***mesh*** y ***surf*** atribuyen color a los bordes y facetas en función de los valores de la función, es decir en función de los valores de la matriz **Z**. Ésta no es sin embargo la única posibilidad. En las siguientes funciones, las dos matrices argumento **Z** y **C** tienen el mismo tamaño:

```
mesh(Z,C)
surf(Z,C)
```

En las figuras resultantes, mientras se dibujan los valores de **Z**, los colores se obtienen de **C**. Un caso típico es aquél en el que se quiere que los colores dependan de la curvatura de la superficie (y no de su valor). MATLAB dispone de la función ***del2***, que aproxima la curvatura por diferencias finitas con el promedio de los 4 elementos contiguos, resultando así una matriz proporcional a la curvatura. Obsérvese el efecto de esta forma de la función ***surf*** en el siguiente ejemplo (si todavía se tiene la matriz **Z** formada a partir de ***test3d***, utilícese. Si no se conserva, vuélvase a calcular):

```
>> C=del2(Z);
>> close, surf(Z,C)
```

9.2.5. FORMAS PARAMÉTRICAS DE LAS FUNCIONES *MESH*, *SURF* Y *PCOLOR*

Existen unas formas más generales de las funciones ***mesh***, ***surf*** y ***pcolor***. Son las siguientes (se presentan principalmente con la funciones ***mesh*** y ***surf***). La función:

```
mesh(x,y,Z,C)
```

dibuja una superficie cuyos puntos tienen como coordenadas (**x(j)**, **y(i)**, **Z(i,j)**) y como color **C(i,j)**. Obsérvese que **x** varía con el índice de columnas e **y** con el de filas. Análogamente, la función:

```
mesh(X,Y,Z,C)
```

dibuja una superficie cuyos puntos tienen como coordenadas (**X(i,j)**, **Y(i,j)**, **Z(i,j)**) y como color **C(i,j)**. Las cuatro matrices deben ser del mismo tamaño. Si todavía están disponibles las matrices calculadas con el fichero ***picos.m***, ejecútese el siguiente comando y obsérvese que se obtiene el mismo resultado que anteriormente:

```
>> close, surf(X,Y,Z), pause(5), mesh(X,Y,Z)
```

¿Cuál es la ventaja de estas nuevas formas de las funciones ya conocidas? La principal es que admiten más variedad en la forma de representar la cuadrícula en el plano (x-y). La primera forma admite vectores **x** e **y** con puntos desigualmente espaciados, y la segunda admite conjuntos de puntos muy generales, incluso los provenientes de ***coordenadas cilíndricas y esféricas***.

9.2.6. OTRAS FUNCIONES GRÁFICAS 3D

Las siguientes funciones se derivan directamente de las anteriores, pero añaden algún pequeño detalle y/o funcionalidad:

surfc	combinación de ***surf***, y ***contour*** en z=0
trisurf	similar a ***surf***, dibuja una superficie 3-D a partir de los valores de una función en una malla de triángulos.
meshz	***mesh*** con plano de referencia en el valor mínimo y una especie de “cortina” en los bordes del dominio de la función
trimesh	similar a ***mesh***, dibuja una superficie 3-D a partir de los valores de una función en una malla de triángulos.
surfl	para controlar la iluminación determinando la posición e intensidad de un foco de luz.
light	crea un foco de luz en los ejes actuales capaz de actuar sobre superficies 3-D. Se le deben pasar como argumentos el color, el estilo (luz local o en el infinito) y la posición. Son muy importantes las propiedades de los objetos iluminados ***patch*** y ***surface***; consultarlas por medio del ***Help*** cuando se vayan a utilizar.
colorbar	añade el mapa de colores activo a la figura, redimensionando los ejes para hacerle un lugar. Se puede colocar horizontal o verticalmente.
sphere	dibuja una esfera 3-D de radio unidad. Por defecto se utiliza un faceteado de 20 (20 meridianos y 20 paralelos). Este número se puede cambiar. Es posible recoger las coordenadas como valor de retorno y multiplicarlas por un factor de escala.
cylinder	dibuja una superficie cilíndrica de radio 1 y altura 1, con 20 facetas laterales. Este número se puede cambiar, como segundo argumento. El primer argumento puede ser un vector que indica como varía el radio en función de la altura del cilindro. También es posible recoger las coordenadas como valor de retorno y multiplicarlas por un factor de escala.

Se pueden probar estas funciones con los datos de que se dispone. Utilícese el ***help*** para ello.

9.2.7. ELEMENTOS GENERALES: EJES, PUNTOS DE VISTA, LÍNEAS OCULTAS, ...

Las funciones ***surf*** y ***mesh*** dibujan funciones tridimensionales en perspectiva. La localización del punto de vista o dirección de observación se puede hacer mediante la función ***view***, que tiene la siguiente forma:

```
view(azimut, elev)
```

donde ***azimut*** es el ángulo de rotación de un plano horizontal, medido sobre el eje **z** a partir del eje **x** en sentido antihorario, y ***elev*** es el ángulo de elevación respecto al plano (x-y). Ambos ángulos se miden ***en grados***, y pueden tomar valores positivos y negativos (sus valores por defecto son -37.5 y 30). También se puede definir la dirección del punto de vista mediante las tres coordenadas cartesianas de un vector (sólo se tiene en cuenta la dirección):

```
view([xd,yd,zd])
```

En los gráficos tridimensionales existen funciones para controlar los ejes, por ejemplo:

```
axis([xmin,xmax,ymin,ymax,zmin,zmax])
axis([xmin xmax ymin ymax zmin zmax cmin cmax])
```

Esta última función es una forma combinada de la función ***axis*** y de la función ***caxis***, explicada en el apartado 9.2.2.

También se pueden utilizar las funciones siguientes: ***xlabel***, ***ylabel***, ***zlabel***, ***xlim***, ***ylim***, ***zlim***, ***axis('auto')***, ***axis(axis)***, etc.

Las funciones ***mesh*** y ***surf*** disponen de un algoritmo de *eliminación de líneas ocultas* (los polígonos o facetas, no dejan ver las líneas que están detrás). El comando ***hidden*** activa y desactiva la eliminación de líneas ocultas.

En el dibujo de funciones tridimensionales, a veces también son útiles los *NaNs*. Cuando una parte de los elementos de la matriz de valores **Z** son *NaNs*, esa parte de la superficie no se dibuja, permitiendo ver el resto de la superficie.